(1)

A NEURAL NETWORK IMPLEMENTATION OF CHAOTIC TIME SERIES PREDICTION

THESIS

James R. Stright
Captain, USAF

AFIT/GE/ENG/88D-50

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GE/ENG/88D-50

A NEURAL NETWORK IMPLEMENTATION OF CHAOTIC TIME SERIES PREDICTION

THESIS

James R. Stright
Captain, USAF

AFIT/GE/ENG/88D-50

AFIT/GE/ENG/88D-50

A NEURAL NETWORK IMPLEMENTATION OF CHAOTIC TIME SERIES PREDICTION

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Electrical Engineering

James R. Stright, B.E.E.
Captain, USAF

December 1988

Acknowledgments

I could not have produced this thesis without the assistance of many people. I am particularly indebted to Dr. Steven K. Rogers, my thesis advisor, who suggested the thesis topic in the first place and thereafter guided me with wisdom and patience along this very new avenue of neural network research. Dr. Dennis Quinn provided several invaluable suggestions at crucial points in my labors. Dr. David Umphress assisted me with the object-oriented design of my predictor network. My fellow students Charles Piazza, Mike Roggemann, Dennis Ruck, and Swen Walker contributed more to my growth as a neural net researcher than any linear combination of sigmoids can express. Thanks, guys.

Finally, the biggest thanks goes to the shortest person, but the one who has borne the heaviest part of this load, my wife Linda.

James R. Stright

Table of Contents

	Page
Acknowledgments	ii
List of Figures	v
List of Tables	vii
Abstract	viii
1. Introduction	1-1
1.1 Historical Background	1-2
1.2 Problem Statement and Scope	1-3
1.3 General Approach	1-3
1.4 Thesis Organization	1-4
2. Background Material	2-1
2.1 Introduction	2-1
2.2 The Glass-Mackey Equation	2-1
2.2.1 General Information	2-1
2.2.2 The Equation and a Typical Representation	2-2
2.3 Chaos and Fractal Dimension	2-4
2.4 An Artificial Neural Network	2-6
2.5 Presentation of Time Series Data to Network	2-11
2.6 Summary	2-13
3. Chaotic Time Series and a Predictor Network	3-1
3.1 Introduction	3-1
3.2 Some Approaches to Solving the Glass-Mackey Eqn.	3-2
3.3 Fractal Dimension of Time Series Data	3-6
3.4 Building a Predictor Network	3-11
3.4.1 Batch Processing with a Degenerate Network	3-11
3.4.2 A Geometric Interpretation of Time Series Learning	3-24
3.4.3 An Object-Oriented Design Predictor Network	3-33
3.5 Summary	3-36

4. Results and Discussion 4-1
4.1 Introduction 4-1
4.2 Sine Wave Results 4-1
4.3 Incommensurate Sine Waves 4-7
4.4 Glass-Mackey Prediction 4-8
4.5 Summary 4-10
5. Conclusions and Recommendations 5-1
5.1 Conclusions 5-1
5.2 Recommendations 5-2
Appendix A: Glass-Mackey Computational Details A-1
Appendix B: The QSD Algorithm B-1
Appendix C: Ada Source Code C-1
Appendix D: Additional Error Surface Descents D-1
Bibliography BIB-1
Vita VIT-1

List of Figures

Figure		Page
2.1	A Chaotic Function x(t)	2-3
2.2	Schematic of a Typical Three Layer Network	2-7
2.3	A Single Neuron	2-8
2.4	The Sigmoid Function f(Z)	2-9
2.5	A Triangle Discriminator Network	2-10
2.6	Plane Region Requiring a Three Layer Network ...	2-11
2.7	Time Series Training and Prediction	2-12
3.1	Comparison of Glass-Mackey Solution Methods	3-3
3.2	Glass-Mackey Solutions with $\Delta t = 1$	3-6
3.3	Glass-Mackey Solutions with $\Delta t = 0.1$	3-6
3.4	Counting Close k-tuples (k = 3)	3-8
3.5	Estimating Fractal Dimension as Linear Slope ...	3-10
3.6	A Sum of Sine Waves	3-11
3.7	Input Plane of a Single Neuron Network	3-12
3.8	Steepest Descent Minimization	3-15
3.9	Cross Sections Containing S_0 and S_1	3-16
3.10	A Simple Error Surface	3-19
3.11	Contour Plot Demonstrating QSD Algorithm	3-19
3.12	Single-Step, Input-by-Input Weight Updating	3-21
3.13	Input-by-Input Application of QSD	3-22
3.14	Single-Step Batch Processing	3-23
3.15	A Simple Time Series	3-25
3.16	Mapping Sequence Pairs to Next Sequence Value ..	3-26
3.17	A Simple Predictor Network	3-26

3.18 Another Time Series 3-29
3.19 Parallel Projections in Two-space 3-30
3.20 A Three Input Predictor Network 3-31
3.21 Booch Diagram of OOD Predictor Network 3-35
4.1 Two Input Sine Wave Prediction 4-3
4.2 Three Input Sine Wave Prediction 4-5
4.3 Incommensurate Sines Prediction 4-8
4.4 Glass-Mackey Prediction 4-10
B.1 Error Surface Contours B-1
B.2 When a Quadratic Approximation Minimizes E B-3
B.3 When a Quadratic Approximation Maximizes E B-6
D.1 QSD Algorithm from (4,-4) D-2
D.2 Single-Step, Input-by-Input Updating from (4,-4) D-2
D.3 Input-by-Input Application of QSD from (4,-4) .. D-3
D.4 Single-Step Batch Processing from (4,-4) D-3
D.5 QSD Algorithm from (6,-8) D-4
D.6 Single-Step, Input-by-Input Updating from (6,-8) D-4
D.7 Input-by-Input Application of QSD from (6,-8) .. D-5
D.8 Single-Step Batch Processing from (6,-8) D-5

List of Tables

Table		Page
4.1	Two Input Sine Wave Prediction	4-2
4.2	Three Input Sine Wave Prediction	4-5
4.3	Incommensurate Sines Prediction	4-8
4.4	Glass-Mackey Prediction	4-9

AFIT/GE/ENG/88D-50

Abstract

This thesis provides a description of how a neural network can be trained to "learn" the order inherent in chaotic time series data and then use that knowledge to predict future time series values. It examines the meaning of chaotic time series data, and explores in detail the Glass-Mackey nonlinear differential delay equation as a typical source of such data. An efficient weight update algorithm is derived, and its two-dimensional performance is examined graphically. A predictor network which incorporates this algorithm is constructed and used to predict chaotic data.

The network was able to predict chaotic data. Prediction was more accurate for data having a low fractal dimension than for high-dimensional data. Lengthy computer run times were found essential for adequate network training.

A NEURAL NETWORK IMPLEMENTATION OF CHAOTIC TIME SERIES PREDICTION

1. Introduction

Accurate prediction of the behavior of complex nonlinear systems has long eluded practitioners of almost every imaginable scientific discipline. Meteorologists have employed models of earth's atmosphere using systems of twelve or more partial differential equations in attempts to predict the weather [Gleick87:19]. Economists have long studied graphs of market trends and Dow Jones averages in attempts to predict market behavior [Gleick87:85]. Physicists have used the world's fastest supercomputers on the nonlinear equations of fluid motion in attempts to track the turbulent flow of a fluid accurately [Gleick87:137]. Such attempts have usually been empirically unsatisfactory, and they have often left scientists with the nagging suspicion that some underlying principle of nonlinear dynamics was eluding them. Addressing the problem of turbulent fluid flow, the quantum theorist Richard P. Feynman noted that it always bothered him that "according to the laws as we understand them today, it takes a computing machine an infinite number of logical operations to figure out what goes on in no matter how tiny a region of space, and no matter how tiny a region of time. How can all that

be going on in that tiny space? Why should it take an infinite amount of logic to figure out what one tiny piece of space/time is going to do? [Feynman65:57]"

1.1. Historical Background

In the past twenty years or so, with insight gained from computer graphics, a new science dubbed Chaos has emerged and has shown considerable promise in explaining complex nonlinear phenomena such as fluid turbulence [Gleick87:202-207]. Although "chaos" has a technical definition, in an intuitive sense, a system is chaotic if it is deterministic but "random" in a manner similar to deterministic pseudo random number generators used on many digital computers [Lapedes88a:1]. Chaos theory tends to discount the long-held assumption that systems which are "visibly unstable, unpredictable, or out of control must either be governed by a multitude of independent components or subject to random external influences. [Gleick87:303]" In chaos theory, the physical source of random-appearing time series data is of little interest for purposes of extending the time series into the future. Instead, chaos predicts future values using only knowledge of the existing data, and a property of the data known as its fractal dimension [Lapedes88a:7].

Fractal dimension may be regarded as a yardstick for measuring along the continuum of order and randomness inherent in dynamical systems (and in time series data, too). Systems with high fractal dimension are highly random

and, from a practical point of view, unpredictable. Systems with low fractal dimension may contain enough order to be predictable [Farmer88:6].

Alan Lapedes and Robert Farber, working at Los Alamos National Laboratory, described their use of a neural network to achieve accurate prediction of a chaotic time series [Lapedes88a].

Neural networks are computational models consisting of one or (usually) more inputs, one or (usually) more interconnected processing elements called neurons, and one or more outputs. They will be discussed more fully in Section 2.4.

This thesis will examine in greater depth the work of Lapedes and Farber, and endeavor to impart a feeling for how a neural network is able to predict.

1.2 Problem Statement and Scope

The general problem of interest is: can a neural network "learn" the underlying order within chaotic time series data and use it to predict future values of the time series accurately? To this end, a suitable batch-processing neural network is constructed and trained to predict chaotic time series data.

1.3 General Approach

Most of the neural networks constructed at AFIT have been used for classification or pattern recognition purposes (see, for example, [Ruck87]). Such networks are said to

employ "value unit coding" because only the relative values from the output nodes are of interest (for example, select the node with the highest value). On the other hand, the numerical value of the output of a predictor network is of direct interest. Such networks are said to employ "variable unit coding". It was felt that this rather unique application warranted development of a unique network, so considerable attention was given to network training algorithms as a prelude to construction of a predictor network.

Neural networks may use any of several algorithms to update the weights and thresholds which they apply to input data. Dahl[87:7] suggests a quadratic method which is elaborated in this thesis, then incorporated into a three layer neural network having only a single output node. Various time series (from simple to complex) are described in terms of the continuous functions from which they are taken. The neural network is trained on early portions of the time series data, then allowed to iterate beyond the time values on which it was trained. The output values are then compared to the known values of the time series to determine the accuracy of the neural network as a predictor.

1.4 Thesis Organization

This chapter has provided a brief historical perspective on chaos, a statement of the goal of this research effort, and an outline of the approach used to perform chaotic time

series prediction. The next chapter will review background material essential to understanding the chaotic time series and the neural network which will be developed in Chapter Three. In Chapter Four, the results of applying time series data to the network will be discussed. Chapter Five will present conclusions and recommendations.

2. Background Material

2.1 Introduction

In the last chapter, a brief historical background of the problem of chaotic time series prediction was presented, as well as a statement of the problem to be solved. This chapter will present background material needed to understand the algorithms which will be developed in Chapter Three and tested in Chapter Four. The following topics will be covered in this chapter: 1) the Glass-Mackey equation, a nonlinear differential delay equation which will be used to represent a deterministically chaotic time series, 2) the concept of fractal dimension as applied to time series data, 3) the architecture and operation of a multilayer perceptron neural network which uses a batch-processing training algorithm, and 4) the iterative method of presenting time series data to the neural network.

2.2 The Glass-Mackey Equation

In this section, the general form of a carefully studied [Farmer82] difference equation is presented, and its chaotic nature is explained.

2.2.1 General Information. A difference equation with a single independent variable, time, specifies the sense in which a dependent variable changes with respect to time at discrete time intervals. A differential equation specifies the sense in which a dependent variable changes continuously with time. Functions described by equations containing both

difference and differential components can appear, when graphed, continuous but only quasi-periodic. Researchers at McGill University in Montreal [Mackey77] found that blood production in the body varies both continuously and discretely with time, there being a discrete time lag between the body's request for more blood and the actual increase in blood production. The name of the Glass-Mackey equation which describes blood production pays tribute to their work.

2.2.2 The Equation and a Typical Representation. In addition to having both differential and delay components, the Glass-Mackey equation is highly nonlinear, having an x^{10} term in the denominator:

$$\frac{dx(t)}{dt} = \frac{a\ x(t-\tau)}{1 + x^{10}(t-\tau)} - b\ x(t) \qquad (2.1)$$

In this equation, x is the function of interest, dependent on the continuous variable time, t. The coefficients a and b are constants, as is the time delay τ days.

Like all differential and difference equations, this one can only be solved with the knowledge of initial conditions which x(t) satisfies. One way to meet this requirement is to assume that x(t) is known for all times between 0 and τ. In this thesis, it is assumed (for computational ease) that x(t) is constant on the interval $[0,\tau]$. Because the function value is specified at an infinite number of points

in $[0,\tau]$, x(t) satisfies in a sense an infinite number of initial conditions.

Following Lapedes, the values of the constants a, b, and τ are taken to be 0.2, 0.1, and 30, respectively [Lapedes88a:5]. Lapedes does not specify the constant he uses to define the initial portion of x(t). If this constant is one, then numerical solution techniques yield the uninteresting function x(t) = 1 for all t (see, for example, Equation (3.2)). Therefore, the constant value two was chosen for x(t) over the interval $[0,\tau]$. With these assumptions, Equation (2.1) can be solved using numerical techniques described in the next chapter. A graph of the solution x(t) over a rather small range of t values is presented in Figure 2.1.

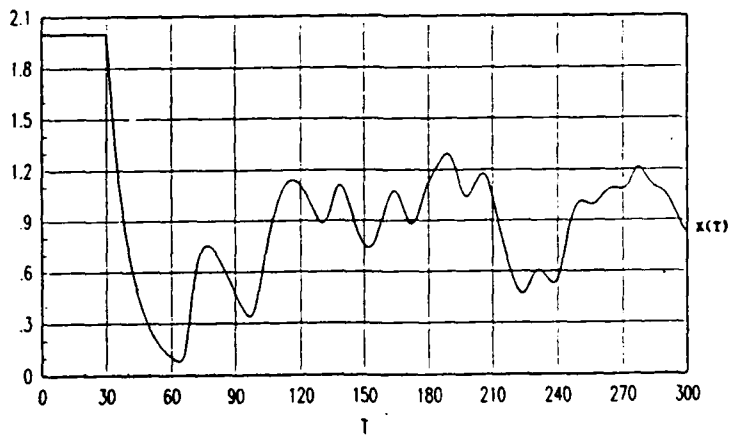


Figure 2.1 A Chaotic Function x(t)

Taking uniform time steps of, for example, $\Delta t = 1$, a time series of data points representing values assumed by

x(t) at times 0, 1, 2, ..., 300 could be tabulated. Without knowledge of the fact that x(t) satisfies Equation (2.1), an observer of the data would be unlikely to find the complicated formula (Equation (3.2)) which generated the time series. He may instead conclude that the data contained no order, in the sense that it was generated from a random, or pseudo random, process, perhaps representing a scaling of Dow Jones closing averages over a 300 day period.

This thesis will show that a neural network is able to "learn" the order inherent in such time series and make accurate predictions of future values.

2.3 Chaos and Fractal Dimension

The fluctuating portion of x(t) in Figure 2.1 appears random. But as Farmer points out [Farmer88:5], randomness is in the eye of the beholder. If randomness is understood to occur to the extent that something cannot be predicted, then perhaps with additional data or closer observation, the data represented in Figure 2.1 can be found to be somewhat orderly, predictable, and hence less random.

Farmer points out that many of the classic examples of randomness are not complicated. The dynamics of flipping a coin, for instance, involve only a few degrees of freedom - the height of the coin from the table on which it will come to rest, the direction and magnitude of the force applied by the thumb - yet the outcomes of coin tosses are considered random. This randomness comes from the very sensitive

dependence of the tosses on the initial conditions. For example, a very small change in the force applied by the thumb can result in a completely different outcome. It is this sensitive dependence which makes prediction difficult.

When sensitive dependence on initial conditions occurs in a sustained way, the system is said to be chaotic [Farmer88:6]. An example is a flag flapping in the wind [Gleick87:5]. The motion of the flag can be described by a position function f and a possibly infinite set of time derivatives $f', f'', \ldots$. The set $\{f, f', f'', \ldots\}$ constitutes a phase space for the system. It turns out that the motion of the flag approaches a subset of the phase space, called an attractor, which has an abstractly defined quantity called its fractal dimension. A small fractal dimension indicates little randomness, and a large fractal dimension indicates great randomness. Chaotic attractors typically have fractal dimensions in the range of 1.95 to 7.5 [Grassberger83:202-203].

Duffing's Equation, $x''(t)+0.1x'(t)+x^3(t) = 12\cos(t)$, which describes the motion of a damped, forced, nonlinear oscillator, provides another example of a chaotic system [Thompson86:7]. The motion x(t) of the oscillator can be drawn as a trajectory in the two-dimensional phase space with coordinate axes x(t) and x'(t). Initial conditions are specified by given values of x(0) and x'(0), and the trajectory formed by increasing the parameter t from 0 falls within a bounded region of phase space. A sampling of this

trajectory at times $t = 2\pi n$, where n is a positive integer, results in an infinite set of distinct points in the phase space. This infinite set is a chaotic attractor for the system. It is an attractor in the sense that other choices of initial conditions result in trajectories which converge to the same set.

The fractal dimension d of the attractor is given by

$$d = \lim_{\epsilon \to 0} \frac{\log [N(\epsilon)]}{| \log(\epsilon) |} \qquad (2.2)$$

where $N(\epsilon)$ is the number of squares with sides of length ϵ needed to completely cover the attractor [Froehling81:607].

Grassberger and Procaccia show how to embed time series data in a "phase" space and extract the fractal dimension of the attractor associated with the data [Grassberger83:200]. Section 3.3 provides the details of their method as applied to the function x(t) of the Glass-Mackey Equation (2.1).

Knowledge of the fractal dimension of time series data is critical to the configuration of the network used to predict it. Lapedes provides an inequality relating the number of inputs required of a predictor network to the fractal dimension of the data supplied [Lapedes88a:6].

2.4 An Artificial Neural Network

Artificial neural networks are models (usually simulated on digital computers) composed of many nonlinear processing elements (called nodes or neurons) operating in parallel and

arranged in patterns reminiscent of biological neuron interconnections [Lippmann87:4]. They have been studied for many years in hopes of achieving human-like performance in the fields of speech and image recognition. Neural networks typically have their nodes arranged in layers, with one input layer, one output layer, and one or more hidden layers between the input and output layers. More than two hidden layers are (in a sense) superfluous [Lippmann87:18], and the network developed in this thesis (see Figure 2.2) has in fact only two hidden layers.

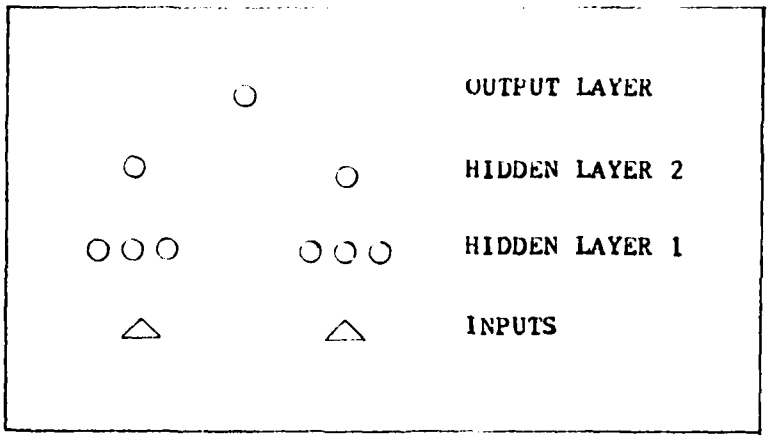


Figure 2.2 Schematic of a Typical Three Layer Network

There is complete interconnection between adjacent layers in the network; for example, each of the six nodes in hidden layer 1 has two inputs.

Each hidden layer node has one variable "weight" for each of its inputs, and exactly one variable "threshold". The neural network is "trained" by presenting to it data in the

form of vectors. If there are N inputs to the network, then it is trained using vectors of N + 1 elements: one element for each input, plus the known (desired) output corresponding to the given input. Training consists of iteratively updating weights and thresholds in the hidden layer nodes as training vectors are repeatedly applied, toward the goal of minimizing the difference between the network's actual and desired outputs.

The operation of a network can be explained in geometric terms, beginning with the operation of a single neuron. Figure 2.3 shows a typical neuron; the weights w_i and threshold θ are imagined residing within the summing node.

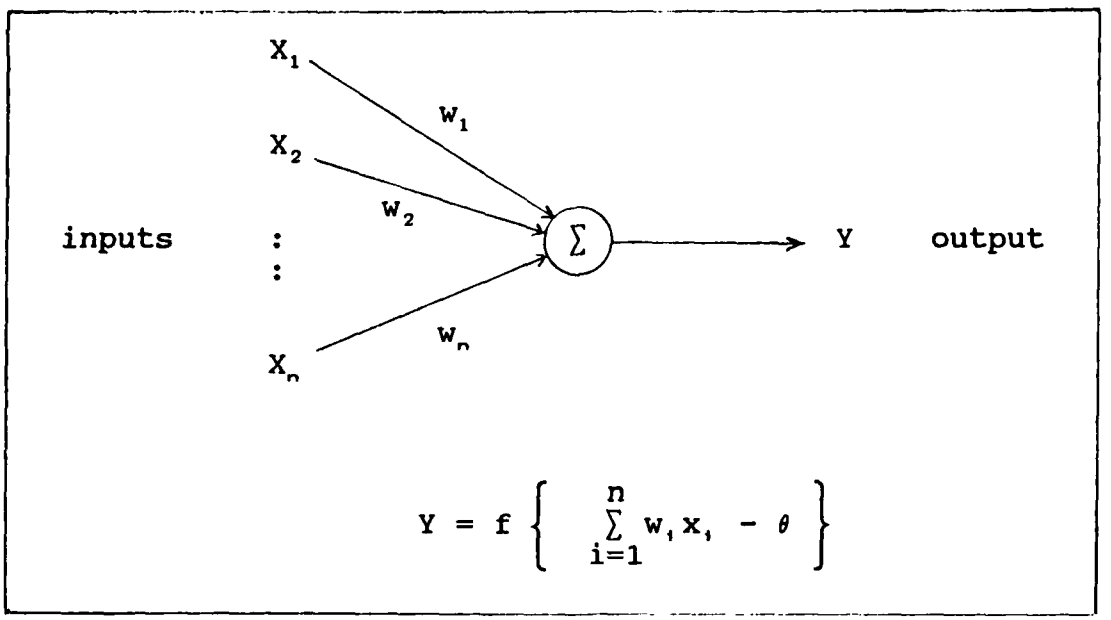


Figure 2.3 A Single Neuron

The output Y of the node is a sigmoid function given by

$$Y = f(Z) = 1/[1 + \exp(-Z)] \qquad (2.3)$$

and is illustrated in Figure 2.4.

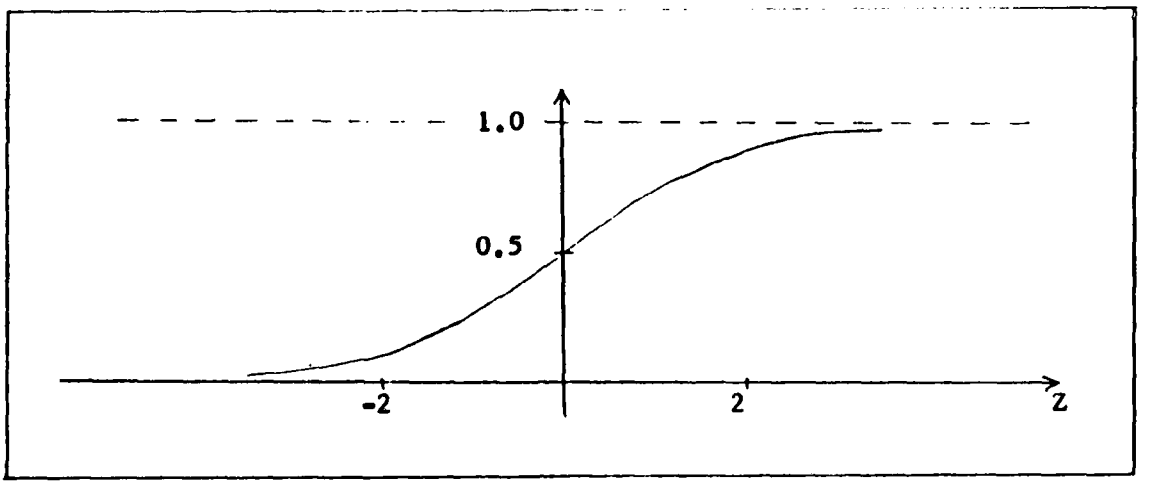


Figure 2.4 The Sigmoid Function f(Z)

The node of Figure 2.3, if presented with two inputs, can be trained to distinguish two adjacent half-plane regions separated by a straight line [Lippmann87:13]. Three such nodes, all receiving the same two inputs, can be trained to distinguish a plane triangular region; see Figure 2.5. That is, this simple network can be trained to respond with an output of one (actually, almost one, since the output has the shape of Figure 2.4) when the coordinates of any point inside the triangle are presented at the network inputs, and respond with an output of zero (actually, almost zero) when the coordinates of any point outside the triangle

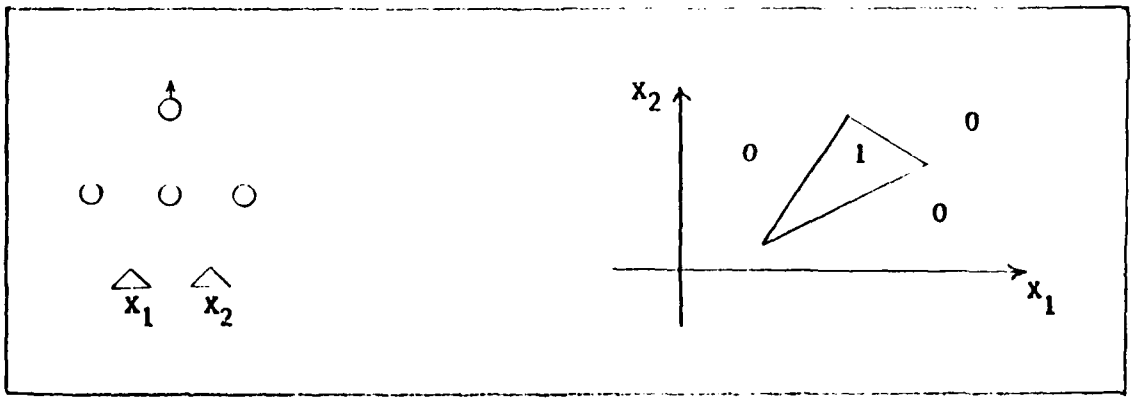


Figure 2.5 A Triangle Discriminator Network

are presented. The accuracy of the network depends on how many points in the plane are used to train the network.

In general, any N-sided convex polygon in the plane can be determined by a three layer network having two inputs, N middle layer nodes, and one output [Lippmann87:16]. Each middle layer node can be considered a participant in a logical AND operation involving one of its half-planes.

To distinguish nonconvex or disjoint planar regions, another middle layer could be added above the existing one. Each node in this layer can be considered a participant in a logical OR operation performed by the output node. This OR operation acts to define the overall region completely. For example, the region illustrated in Figure 2.6 requires the use of two nodes in the layer immediately below the output node. The network of Figure 2.2 could accomplish this.

This research will use a three layer network for time series prediction. An explanation of the network, including its weight update algorithm, is presented in Chapter 3.

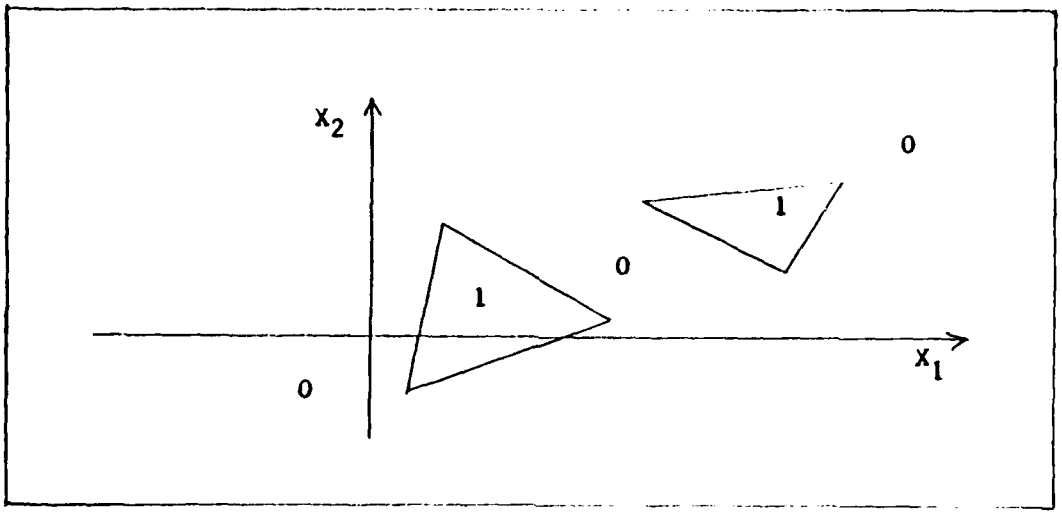


Figure 2.6 Plane Region Requiring a Three Layer Network

2.5 Presentation of Time Series Data to Network

Three layer neural networks are not limited to applications involving the recognition of geometric regions. A three layer network can "learn" the order inherent in a time series, then use that "knowledge" to predict later time series values by feeding its delayed output into an input node.

The number m of network inputs needed for accurate prediction must satisfy

$$d < m + 1 < 2d + 1 \qquad (2.4)$$

where d is the fractal dimension of the time series data [Lapedes88a:6].

Suppose a finite sequence of M time series values S_1, $S_2, \ldots, S_M$ are spaced equally in time, and it is desired to predict series values for times larger than M. The approach

taken is illustrated in Figure 2.7 with a network having four inputs. The network is trained using the 5-ary training vectors $(S_1, S_2, S_3, S_4, S_5)$, $(S_2, S_3, S_4, S_5, S_6)$, ... , $(S_{M-4}, S_{M-3}, S_{M-2}, S_{M-1}, S_M)$. When training is complete, all network weights are fixed and the network output is a complicated but deterministic function of the network inputs. The value S_{M+1} is then predicted as the network output corresponding to the input vector $(S_{M-3}, S_{M-2}, S_{M-1}, S_M)$. The value S_{M+1} then serves as the last component of the next input vector, which is used to predict S_{M+2}. The prediction process continues as long as desired.

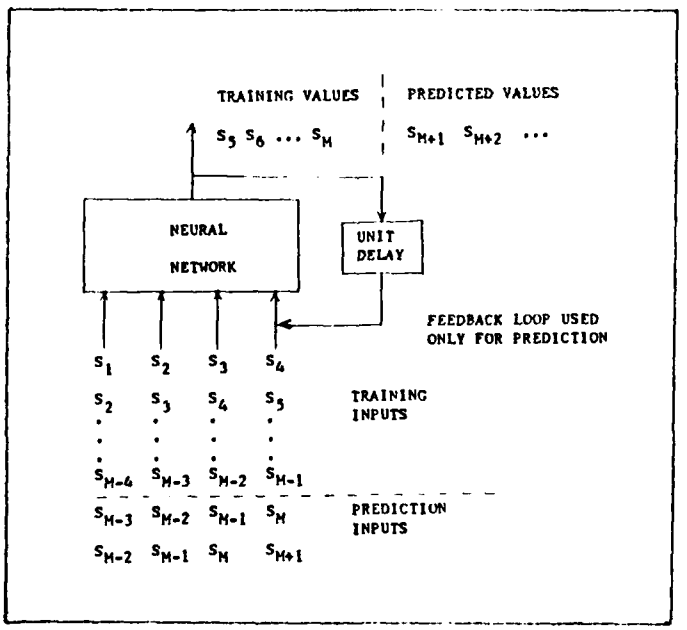


Figure 2.7 Time Series Training and Prediction

Prediction accuracy generally decreases with time due to incomplete training and numerical rounding.

2.6 Summary

This chapter has reviewed work done on an important source of chaotic time series data, the Glass-Mackey equation. It has described the related concepts of fractal dimension and chaos. Neural network operation has been explained in general terms, and a predictor network configuration was given. The next chapter will show how chaotic data can be generated, examine some network training algorithms, and explain how a predictor network can be constructed.

3. Chaotic Time Series and a Predictor Network

3.1 Introduction

The chaotic natures of several standard dynamical systems have been studied extensively in the last ten years or so [Grassberger83:193]. Usually these systems involve several independent variables. Although they occur frequently in nature [Lapedes88a:5], chaotic systems which involve a single function of a single independent variable have been less intensely investigated. Certainly the best documented system of this type is the model of blood production studied by Mackey and Glass in the mid '70s [Mackey77:287]. The system gives rise to a nonlinear differential delay equation which bears their names (Equation (2.1)). The solution to this equation represented an ideal source of chaotic data for this thesis, although far better prediction results were obtained using data from a simpler nonperiodic function (which is described in Section 3.3).

This chapter will examine how the Glass-Mackey equation can be solved. The fractal dimension of a typical example will be determined. Using the same method, the fractal dimension of a simpler nonperiodic function (a sum of sinusoids with incommensurate frequencies) will also be found. Attention will then be turned to the neural network which will endeavor to predict chaotic function values after invocation of a suitable training algorithm.

3.2 Some Approaches to Solving the Glass-Mackey Equation

The Glass-Mackey equation, Equation (2.1), does not have a closed form solution. However, two techniques of numerical analysis readily yield approximate solutions. The first, Euler's Method [Conte72:329], approximates future values x(t) based on the definition of the derivative of a function. If a time step Δt is sufficiently small, it is reasonable to make approximations directly from Equation (2.1):

$$\frac{dx(t)}{dt} = \frac{a\, x(t-\tau)}{1 + x^{10}(t-\tau)} - b\, x(t)$$

$$\approx \text{the slope of } x(t) \text{ at } t$$

$$\approx \frac{x(t + \Delta t) - x(t)}{\Delta t}$$

Solving for $x(t + \Delta t)$ yields

$$x(t + \Delta t) \approx x(t) + \Delta t \frac{dx(t)}{dt} \triangleq x_1(t + \Delta t)$$

which gives future values of x(t) entirely in terms of past values. This Euler's method solution is labeled $x_1(t)$ in Figure 3.1, where the initial value of x(t) is 2 on the interval from 0 to $\tau = 30$, and where $a = 0.2$, $b = 0.1$, and $\Delta t = 3$.

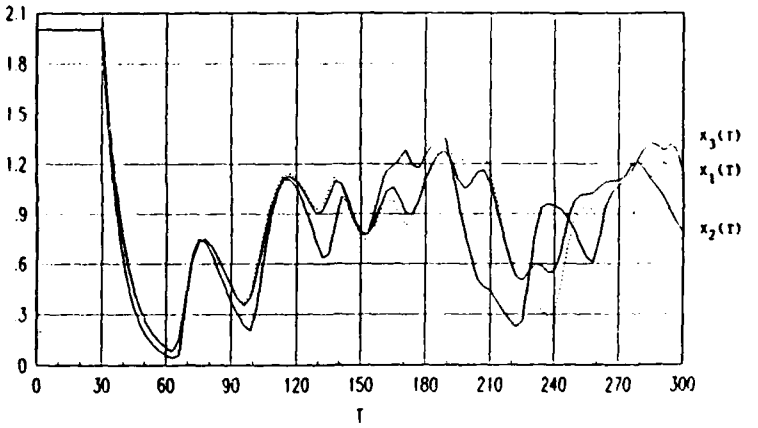


Figure 3.1 Comparison of Glass-Mackey Solution Methods

The predictor-corrector method [Conte72:347] can also provide a Glass-Mackey solution. Define f(t) by

$$f(t) = \frac{dx(t)}{dt} = \frac{a\ x(t-\tau)}{1 + x^{10}(t-\tau)} - b\ x(t)$$

so that

$$\int_{t}^{t+\Delta t} f(\alpha)\ d\alpha = \int_{t}^{t+\Delta t} \frac{dx(\alpha)}{d\alpha}\ d\alpha$$

The integral on the left can be approximated by the area of a trapezoid, and the integral on the right evaluates simply by the fundamental theorem of integral calculus. The last equation therefore becomes

$$\frac{\Delta t}{2} \left\{ f(t + \Delta t) + f(t) \right\} \approx x(t + \Delta t) - x(t) \qquad (3.1)$$

This equation is solved for $x(t + \Delta t)$ in Appendix A, giving the following expression for the predictor-corrector approximation:

$$x_2(t + \Delta t) = \frac{2 - b\,\Delta t}{2 + b\,\Delta t}\, x(t)$$

$$+ \frac{\Delta t}{2+b\Delta t} \left\{ \frac{ax(t-\tau)}{1+x^{10}(t-\tau)} + \frac{ax(t+\Delta t-\tau)}{1+x^{10}(t+\Delta t-\tau)} \right\} \qquad (3.2)$$

The predictor-corrector solution is labeled $x_2(t)$ in Figure 3.1.

The predictor-corrector solution is an improvement over Euler's method [Conte72:347], but without going into an in-depth analysis of the error terms at each iteration, it is difficult to get a feeling for the absolute accuracy of the solution. Instead, a third approach to solving Equation (2.1) is possible, one which gives a solution which converges to the predictor-corrector solution even for relatively large values of Δt. This fact allows the use of the predictor-corrector solution as a baseline with a high degree of confidence.

This third approach involves multiplying the Glass-Mackey equation by an exponential integrating factor and again applying the trapezoidal rule to an integration. Although the integrating factor technique is a classical method, it was suggested for this application by Dr. Dennis Quinn of the mathematics department at AFIT. The derivation of the

approach is provided in Appendix A. To a good approximation, at a time $\tau + k\,\Delta t$ (where k is a positive integer) the function x(t) is given by

$$x_3(\tau + k\,\Delta t) = x(\tau)\exp(-bk\Delta t) + \Delta t \exp[-b(\tau+k\Delta t)]\{0.5G(\tau) + G(\tau+\Delta t) + \cdots + G[\tau+(k-1)\Delta t] + 0.5G(\tau+k\Delta t)\} \quad (3.3)$$

where $G(s) = ax(s-\tau)\exp(bs) / [1 + x^{10}(s-\tau)]$.

The integrating factor method is labeled $x_3(t)$ in Figure 3.1. The three estimates x_1, x_2, and x_3 in this figure are based on $\Delta t = 3$, which is one-tenth the length of the initial $[0,\tau]$ interval. In Figures 3.2 and 3.3, the size of Δt is reduced progressively to values of 1 and 0.1. The apparent convergence of x_2 and x_3 support the choice of either method as a "correct" representation of x(t). However, the integrating factor method which generates $x_3(t)$ experiences numerical failures for large values of t (or equivalently, for large values of k in Equation (3.3)) due to the presence of exponential terms. Therefore, the "correct" x(t) used in this thesis was taken as x_2. It generated a time series 350 τ-units in length, taking time steps $\Delta t = 0.1$. This means that the domain of x(t) actually extends from 0 to 10,500 instead of only to 300 as depicted in Figures 3.1 through 3.3. Fortunately, time samples for predicting this chaotic time series are required only at intervals of about t = 6 days [Lapedes88a:6].

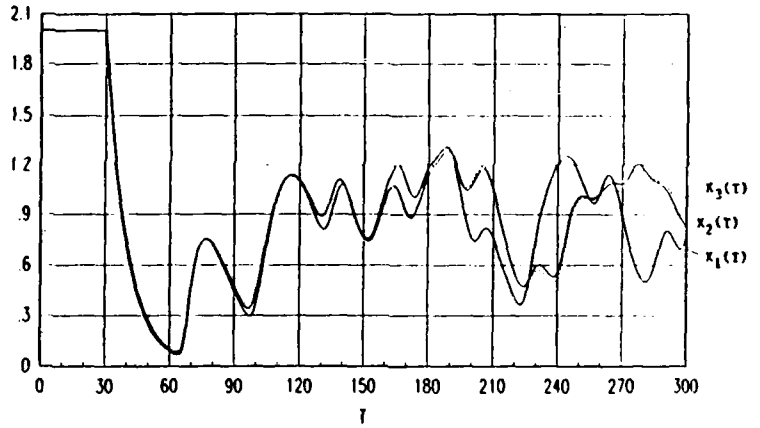


Figure 3.2 Glass-Mackey Solutions with $\Delta t = 1$

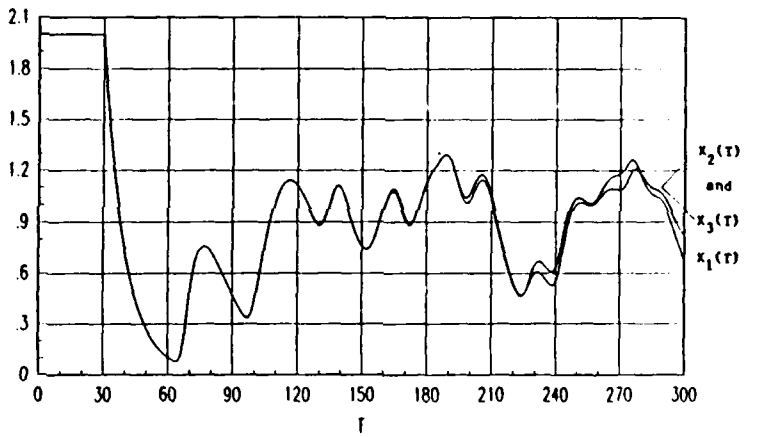


Figure 3.3 Glass-Mackey Solutions with $\Delta t = 0.1$

3.3 Fractal Dimension of Time Series Data

The preceding section showed how the chaotic time series data was generated. As explained in Section 2.5, this data was used as input to a neural network which was used to predict future time series values. To be an effective predictor, the number of network inputs depends on a property of the time series data known as its fractal

dimension [Lapedes88a:6]. Specifically, if d represents the fractal dimension of the data and m represents the number of network inputs, then m must satisfy $d < m+1 < 2d+1$.

Roughly speaking, the fractal dimension of the data is a measure of the "raggedness" of the graphed data. Very smooth data has a low fractal dimension (eg, a straight line has a fractal dimension of 1.0) and chaotic data has a high fractal dimension (typically in the range of 1.95 to 7.5). Grassberger and Procaccia [Grassberger83] describe and justify a practical method for determining the fractal dimension of chaotic time series data. This thesis utilizes their method.

This method first makes a k-dimensional space consisting of k-tuples taken from the time series. Here k is an estimate chosen slightly larger than the actual fractal dimension being sought; a rough estimate ($d < k < d+6$) is sufficient [Grassberger83:200]. A set of small numbers l_i (about six or seven is adequate) is chosen, and for each i the number of pairs of contiguous k-tuples within Euclidean distance l_i of each other is determined and denoted $C(l_i)$. Figure 3.4 illustrates a case where the 3-tuple (P_M, P_{M+1}, P_{M+2}) is close to (P_0, P_1, P_2) in Euclidean distance and would likely contribute to the count $C(l_i)$ for most l_i. On the other hand, (P_5, P_6, P_7) is relatively far from (P_0, P_1, P_2) and might not contribute to the count $C(l_i)$ for any of the selected l_i. The Grassberger algorithm can be implemented by taking the smallest l_i and the leftmost 3-

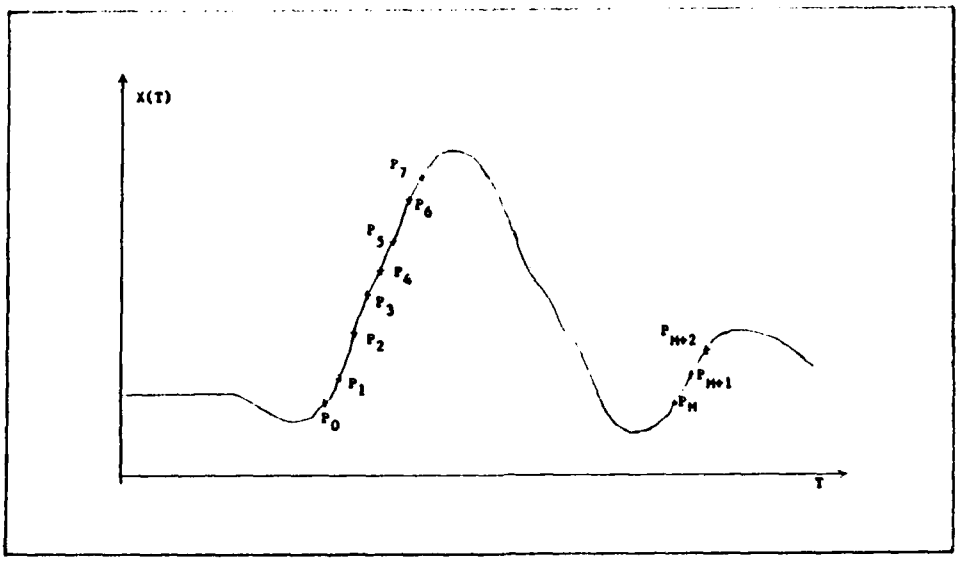


Figure 3.4 Counting Close k-tuples (k = 3)

tuple (P_0, P_1, P_2) and checking each 3-tuple to the right for its adherence to the l_1 distance criterion - first (P_1, P_2, P_3), then (P_2, P_3, P_4), continuing until all contiguous 3-tuples are compared with (P_0, P_1, P_2). After the rightmost 3-tuple has been compared, the process repeats beginning at the left with (P_1, P_2, P_3). The $C(l_1)$ counter continues to grow as more pairs of 3-tuples are found to fall within the allowable l_1 distance. After the rightmost 3-tuple is compared with (P_1, P_2, P_3), (P_2, P_3, P_4) serves as the leftmost 3-tuple for the next iteration.

Continuing in like manner, leftmost finally becomes rightmost, and the counter $C(l_1)$ indicates the total number of contiguous 3-tuples which lie within distance l_1 of each other. The numbers l_1 and $C(l_1)$ are stored, another l_j is selected, and the process begins anew with (P_0, P_1, P_2) at the

left. It terminates with $C(l_j)$ determined. All pairs $(l_n, C(l_n))$ are likewise determined and stored.

Grassberger and Procaccia show that if the data used is chaotic rather than random, arising from an attractor of dimension less than about seven, and if a large enough number of sample points are used (roughly five thousand is usually adequate), then the pairs obtained by this algorithm display a logarithmic linearity (or near linearity). That is, if the pairs $(\log(l_i), \log(C(l_i))$ are plotted, they are found to be collinear. The slope of this line is a good approximation of the fractal dimension of the data.

As a computational convenience, a constant scaling factor of l_0 may be introduced, so that $\log(C(l_i))$ is plotted against $\log(l_i/l_0)$. This does not affect the linearity of the plot, since $\log(l_i/l_0) = \log(l_i) - \log(l_0)$. Figure 3.5 shows plots for four k values based on the Glass-Mackey data described in Section 3.2. The l_i values were 0.1, 0.2,..., 0.7, and l_0 was 0.1. Notice that all four sets of data points are nearly collinear, although the collinearity seems to increase somewhat as k increases. A reasonable estimate of the fractal dimension d of the Glass-Mackey data can be determined as the slope of any of these lines; from k = 7, conclude that $d \approx 2.5$.

The criterion $d < m+1 < 2d+1$ (Equation (2.4)) in this case indicates that the number m of network inputs should be between 1.5 and 5. In this research, m = 4 was used,

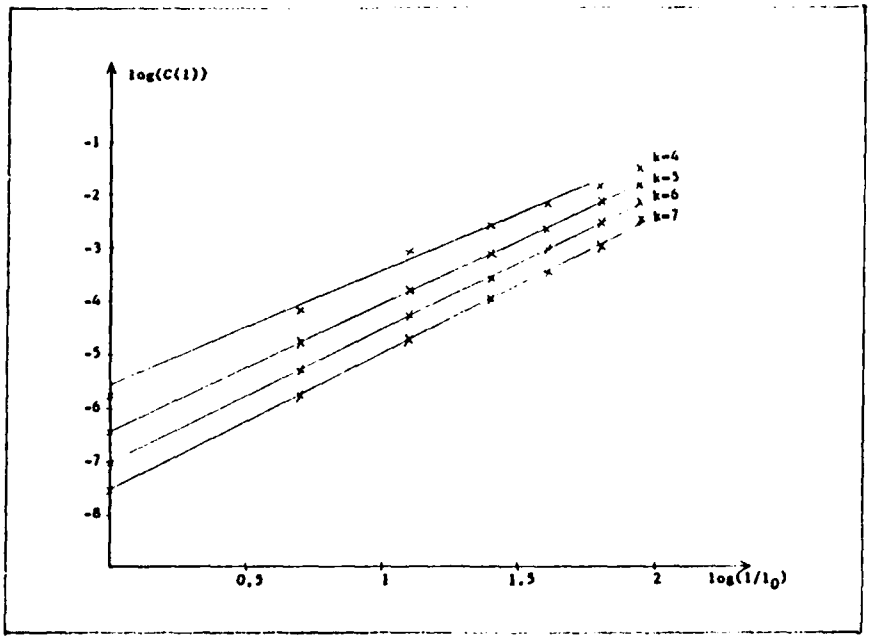


Figure 3.5 Estimating Fractal Dimension as Linear Slope

following Lapedes' work with similar data having fractal dimensions of 2.1 and 3.5 [Lapedes88b:15].

Lapedes mentions that the results he obtained in predicting Glass-Mackey data involved training run times of 30 to 60 minutes on a Cray X-MP computer [Lapedes88b:19]. The research conducted for this thesis used a VAX 11/780 as the primary host. It was hoped that a time series which was somewhat less random would prove manageable on the VAX while lending insight to network parameter values which might allow some degree of Glass-Mackey prediction. The function chosen was a sum of sine waves with incommensurate frequencies (frequencies are incommensurate if their ratio is an irrational number). Specifically, the function was

$$y(t) = [2 + \sin(2^{1/2}\ t) + \sin(3^{1/2}\ t)]/2$$

and its fractal dimension was also determined using the Grassberger-Procaccia method. A data set of 5220 samples of y(t) (taken at intervals of $\Delta t = 2$) were input to the same fractal dimension program used to generate the data of Figure 3.5, yielding a fractal dimension of 1.7. A graph of a small sample of y(t) is shown in Figure 3.6. Like the Glass-Mackey solution, it is nonperiodic.

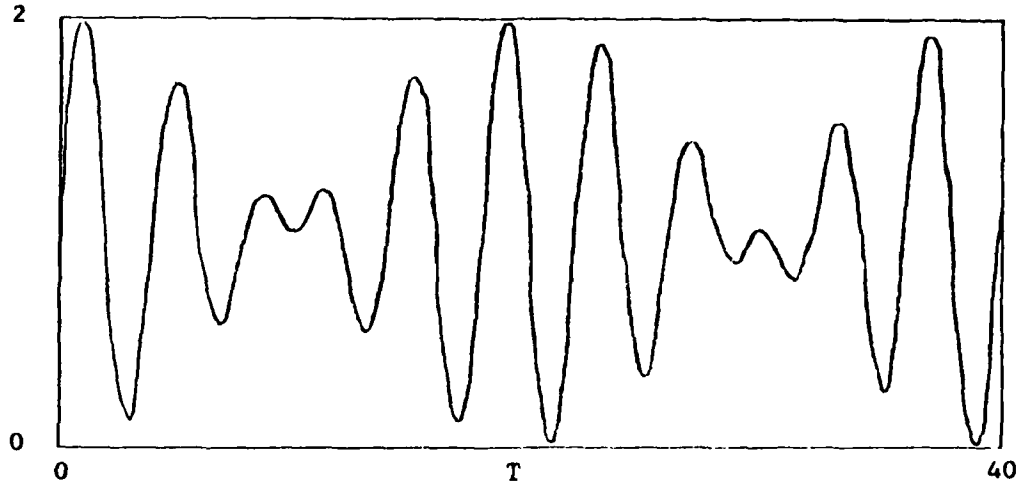


Figure 3.6 A Sum of Sine Waves

3.4 Building a Predictor Network

3.4.1 Batch Processing with a Degenerate Network. The simplest possible neural network consists of a single neuron. Valuable insights into the behavior of large networks follow from a careful examination of their most basic building blocks.

A single neuron with two inputs can be trained to distinguish any two half-plane regions determined by a

straight line [Lippmann87:13]. Figure 3.7 shows the line $X_2 = X_1$ and four points which will be used to train a single

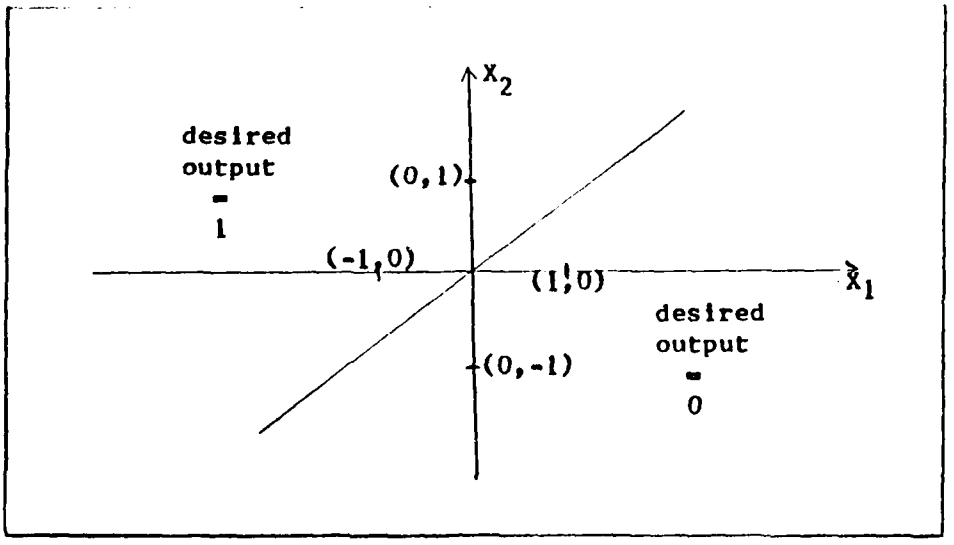


Figure 3.7 Input Plane of a Single Neuron Network

neuron to approximate this line. Using the notation of Figure 2.3, this simple network has only two inputs X_1 and X_2, and two corresponding weights W_1 and W_2. The threshold θ is given the constant value zero, and the output is assumed sigmoidal so $Y = 1/[1 + \exp(-W_1 X_1 - W_2 X_2)]$. The choice of $\theta = 0$ makes Y a simple two dimensional function of weight space, so that an error function may be graphically represented above the weight space. The network is to be trained in such a way that points in the upper left region of the input plane (including the points (0,1) and (-1,0)) produce outputs $Y \approx 1$, and points in the lower right region produce outputs $Y \approx 0$. Although a sigmoid never actually attains the values 1 and 0, these are nevertheless referred

to as the desired outputs within their respective half-plane regions. Thus training consists of finding values for the weights W_1 and W_2 which result in a three dimensional sigmoid above the input plane. The sigmoid is given by $Y = 1/[1 + \exp(-W_1X_1 - W_2X_2)]$; it is nearly flat and zero-valued in the lower right region, it rises sharply to values near one-half at the line $X_2 = X_1$, and is nearly flat and unity-valued in the upper left region. It appears in cross section in Figure 2.4.

Because a sigmoid only approaches zero and one asymptotically, any choice of values for W_1 and W_2 will yield Y values which differ at least slightly from the desired outputs, regardless of location in the input plane. The best training can accomplish is to minimize these differences, or errors, based on the known desired values for known inputs. One way to do this is to construct a positive-valued error function which has W_1 and W_2 as its only independent variables and which incorporates all known input coordinates and desired outputs. This approach is called batch processing of input vectors (an N-tuple of input coordinates, followed by a desired output value, constitute an N + 1 dimensional "input vector").

Lapedes[88a:3] suggests using a positive additive contribution from each input vector in constructing an error function. In the present example, suppose a sigmoid $Y(X_1,X_2;\omega_1,\omega_2) = 1/[1 + \exp(-\omega_1X_1 - \omega_2X_2)]$, with ω_1 and ω_2 fixed, is suggested as a solution to the half-plane

separation problem of Figure 3.7. Any notion of the error associated with this solution must be based on the errors arising at each known set of input coordinates. A reasonable approach is to sum the squares of the errors at each set of known inputs (the squares are used to avoid the possibility of negative contributions to the total error). Thus the error $E(\omega_1,\omega_2)$ for the solution $Y(X_1,X_2;\omega_1,\omega_2)$ is given by

$$\begin{aligned} E(\omega_1,\omega_2) = \; & [\text{desired}(-1,0) - Y(-1,0)]^2 \\ & + [\text{desired}(0,1) - Y(0,1)]^2 \\ & + [\text{desired}(1,0) - Y(1,0)]^2 \\ & + [\text{desired}(0,-1) - Y(0,-1)]^2 \\ = \; & [1 - Y(-1,0)]^2 + [1 - Y(0,1)]^2 \\ & + [0 - Y(1,0)]^2 + [0 - Y(0,-1)]^2 \end{aligned} \tag{3.4}$$

This numerical value may now be compared to other values of error arising from other potential solutions $Y(X_1,X_2;w_1,w_2)$, and the pair of weights (w_1,w_2) which has the minimum error value $E(w_1,w_2)$ may be thought of as the best weight-space solution of all pairs considered.

In general, the batch-processing error function of any single-output network may be expressed as

$$E(\underline{W}) = \sum_{\underline{s}\in S} [\text{desired}(\underline{s}) - \text{actual}(\underline{s})]^2 \tag{3.5}$$

where $\underline{W}$ is a vector of all variable network weights and S is the set of network training vectors. The goal of any

training algorithm is to find a point in weight space which in some sense minimizes the value of the function E.

Suppose a point $\underline{W}(0)$ is chosen more or less at random in weight space, and for convenience assume $\underline{W}(0)$ is in two-space (as in the half-plane separation problem). Figure 3.8 depicts the initial steps of a "steepest descent" method of finding the minimum value of the error surface E, which is assumed to lie at the center of the concentric contours.

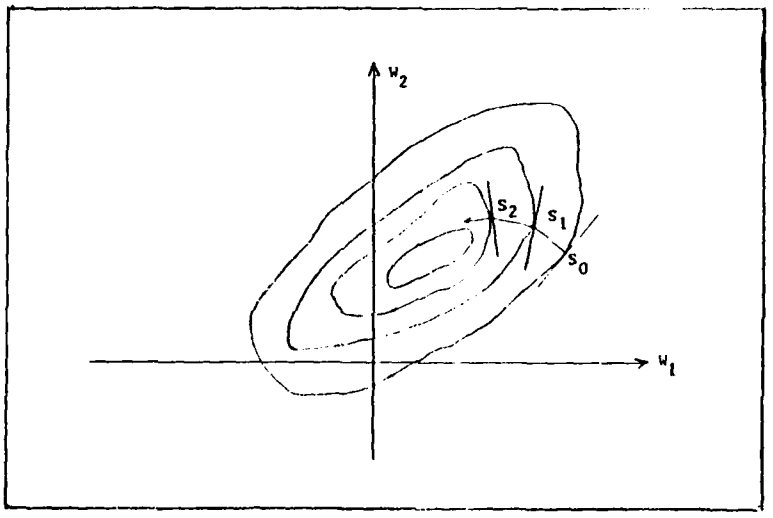


Figure 3.8 Steepest Descent Minimization

Points S_0, S_1, and S_2 lie on the surface above the points $\underline{W}(0)$, $\underline{W}(1)$, and $\underline{W}(2)$, respectively. First the gradient of E (denoted ∇E) is determined at $\underline{W}(0)$. The gradient is a vector which points in the direction of greatest change of E and has a magnitude proportional to the steepness of E. The gradient at $\underline{W}(0)$ lies in a unique plane that is itself perpendicular to the (W_1, W_2) plane. This plane is shown in two orientations in Figure 3.9 where it is understood that

the horizontal axis A lies in the (W_1, W_2) plane, and P_0 and P_1 correspond to $\underline{W}(0)$ and $\underline{W}(1)$ respectively. In this case

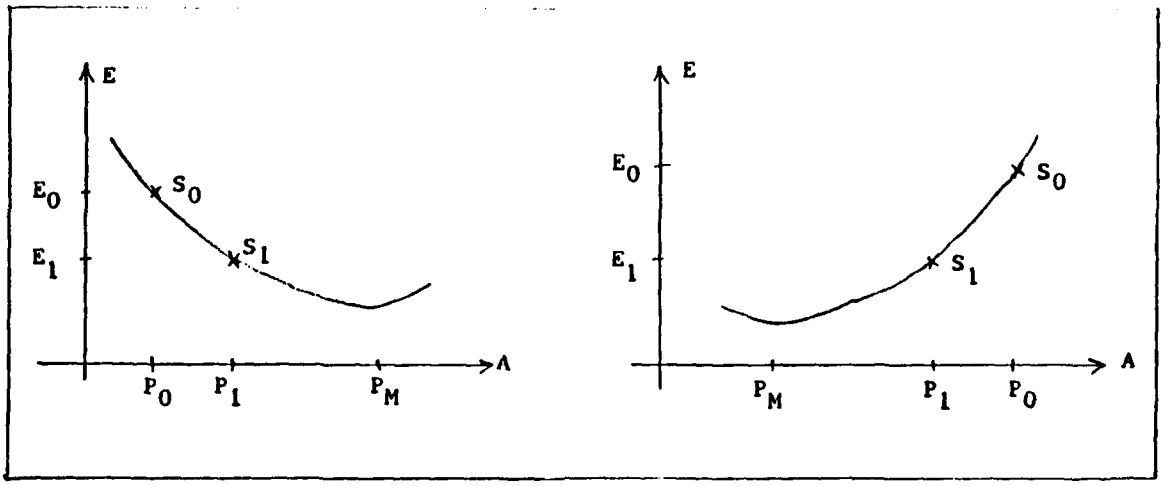


Figure 3.9 Cross Sections Containing S_0 and S_1

the error function may be expressed as E(A), a function of the single variable A. A new point $\underline{W}(1)$ in weight space is found using the rule $\underline{W}(1) = \underline{W}(0) - \eta \nabla E(\underline{W}(0))$, where η is a small positive value. This rule ensures that $\underline{W}(1)$ will lie on the horizontal A axis.

Points P_1 on the A axis may also be generated by the rule

$$P_1 = P_0 - \eta \frac{dE}{dA}(P_0) \qquad (3.6)$$

In either case, as η becomes small, $\underline{W}(1)$ becomes close to $\underline{W}(0)$ and the point P_1 approaches P_0. Separating the components of the vector equation, this may also be written as the pair of equations

$$W_1(1) = W_1(0) - \eta \frac{\partial E}{\partial W_1}(\underline{W}(0))$$

$$W_2(1) = W_2(0) - \eta \frac{\partial E}{\partial W_2}(\underline{W}(0))$$

If the step size η is chosen small enough, the value E_1 of E at the new point S_1 will be smaller than the value E_0 of E at the original point S_0. To see this, suppose the direction of the A axis in Figure 3.9 had been chosen so that E is sloping down to the right at S_0. Then the derivative of E with respect to A is negative at P_0. Since E is a smooth function, this means that this derivative must remain negative in some neighborhood of P_0.

Choosing η small enough will ensure that the point

$$P_1 = P_0 - \eta \frac{dE}{dA}(P_0)$$

lies within this neighborhood. Since $\eta > 0$, this means $P_1 > P_0$ and consequently $E(P_1) < E(P_0)$.

On the other hand, if E is sloping up to the right in Figure 3.8, then the derivative of E with respect to A is positive at P_0. In this case, for small enough η,

$$P_1 = P_0 - \eta \frac{dE}{dA}(P_0) < P_0$$

and again $E(P_1) < E(P_0)$.

Having thus arrived at the point S_1 on the error surface of Figure 3.8, the same process may be applied to yield the point S_2, and so forth until a suitable weight update termination criterion is met. However, Dahl[88:7] suggests using in a neural network an update method which fits a parabolic approximation to each cross-sectional slice of the error surface and iterates using the minima of these parabolas. Instead of using point P_1 in Figure 3.9 to determine the first point of iteration S_1, this method uses an approximation to the point P_M. Appendix B contains the details of this method. Although it is a classical minimization technique, it is here called the QSD algorithm for convenience. It is the method used to update all variable weights in the predictor network developed in this thesis.

An exploration of the error surface arising from the simple half-plane separation network shows the advantage of this technique. Figure 3.10 is a three dimensional plot of this surface as calculated directly from Equation (3.4). Figure 3.11 gives a contour representation of the same surface. Superimposed on this surface is a sequence of vectors showing each jump taken by the QSD algorithm, beginning at the initial weight pair (4,-3.5). The W_1 and W_2 axes both extend from -10 to 10. The lowest contour line is labeled 1, and the highest is labeled 30. The length of each vector is represented by the length of its arrowhead. The algorithm terminates when the distance between

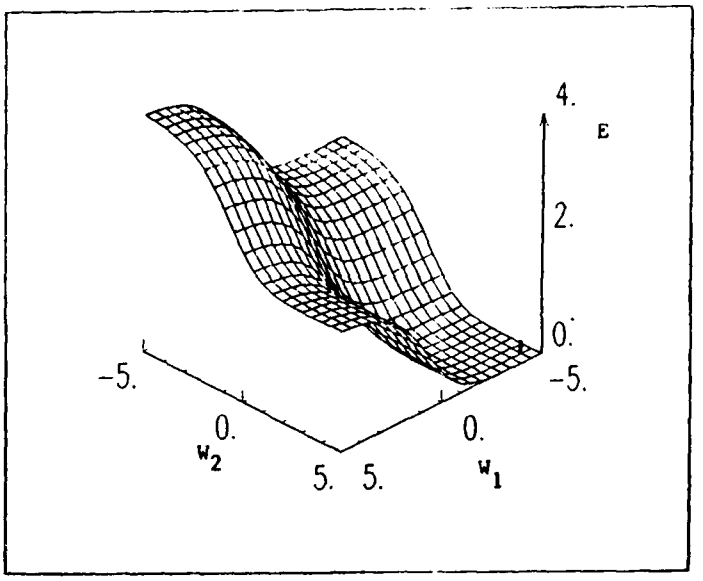


Figure 3.10 A Simple Error Surface

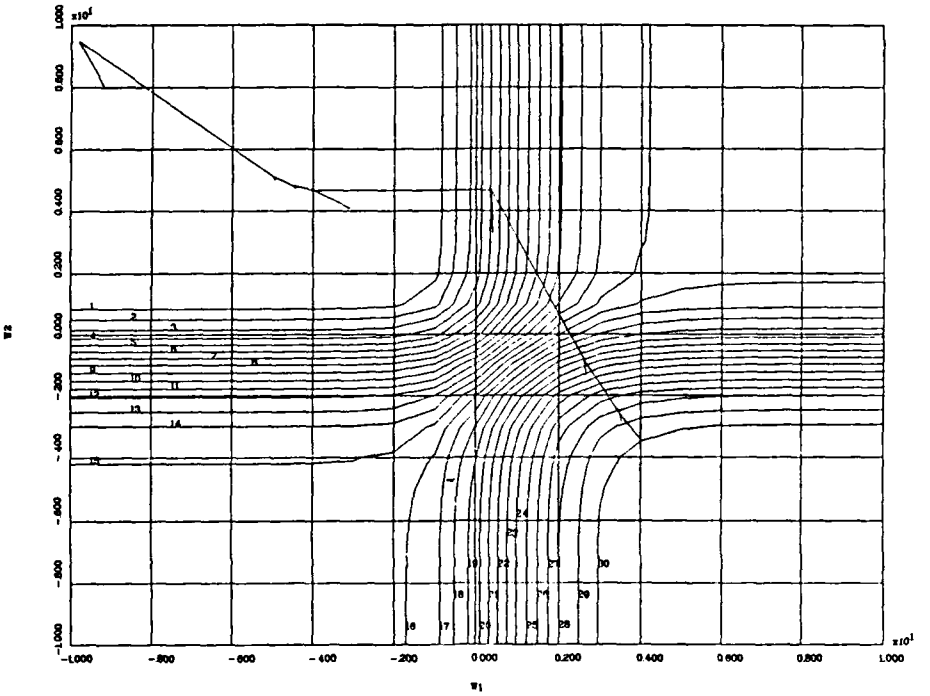


Figure 3.11 Contour Plot Demonstrating QSD Algorithm

successive weight pairs is less than 0.00065 (chosen by trail and error), or when 240 weight updates have been performed, whichever occurs first. Figure 3.11 reveals that the QSD algorithm terminated after 7 weight updates at the weight pair $(W_1, W_2) = (-9.8, 9.4)$.

Compare this performance to that of an update algorithm which changes weights on the basis of one training vector per update [Lippmann87:17]. The weight update rule in this case reduces to $P_1 = P_0 - \eta \nabla E(P_0)$ at each iteration, where $E(\underline{W}) = [\text{desired}(\underline{s}) - \text{actual}(\underline{s})]^2$ for a single input vector $\underline{s}$. No attempt is made to seek a lower point on the plane determined by $\nabla E(P_0)$, and $\nabla E(P_0)$ is based on only a single component of the error surface defined in Equation (3.5). Figure 3.12 shows the evolution of weights in such an algorithm. The parameter η (corresponding to the initial step size A_1 of Appendix B) and criteria for update termination are the same as those applicable to the QSD algorithm depicted in Figure 3.11. Hundreds of small updates occur in Figure 3.12 to the left of $W_1 = -2.0$. The algorithm actually terminates at $(W_1, W_2) = (-4.0, 4.0)$ after the maximum allowed number of iterations, 240.

The QSD technique of minimizing cross-sections can also be applied on an input-by-input basis, though generally with less satisfactory results than obtainable with batch processing. Figure 3.13 shows weights being updated from the same initial point and with the same parameters as before. As in Figure 3.12, updating causes weights to be

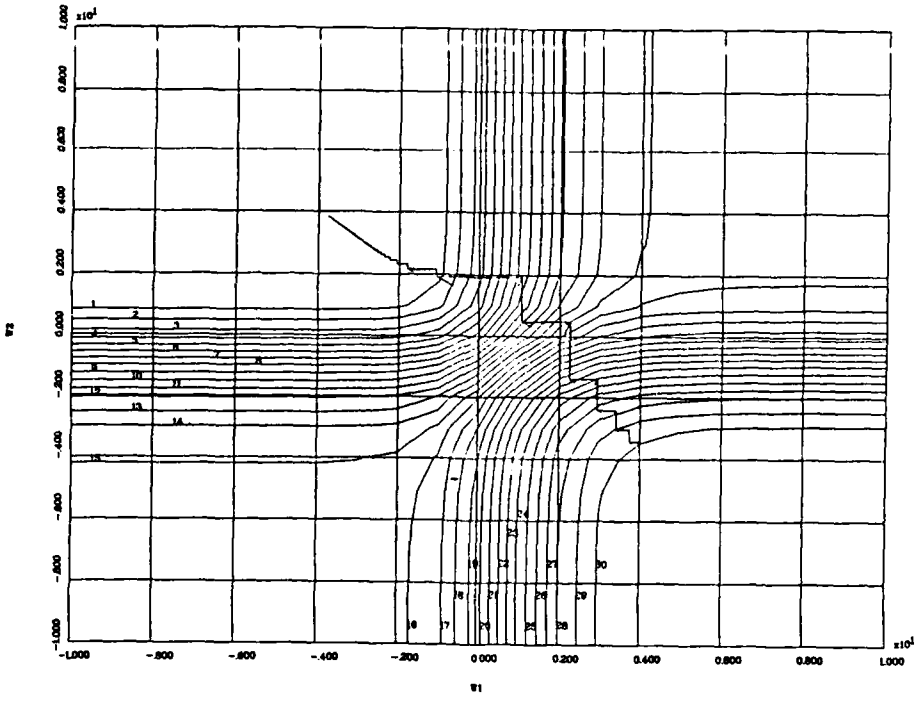


Figure 3.12 Single-Step, Input-by-Input Weight Updating

shifted in rectilinear fashion through weight space. This is because each contributing component of the total error function has only one nonvanishing weight component. For example, the last squared term of Equation (3.4) has a zero coefficient with its W_1 component: $Y(0,-1) = 1/[1 + \exp(0 \cdot W_1 - W_2)]$. The square of $Y(0,-1)$ is also roughly sigmoidal and varies only in the W_2 direction.

The final weight update algorithm to be considered avoids rectilinear updating, as does the QSD algorithm, by constructing the total error surface according to Equation (3.4) and descending according to the gradient of the surface. This makes it, too, a batch processing algorithm.

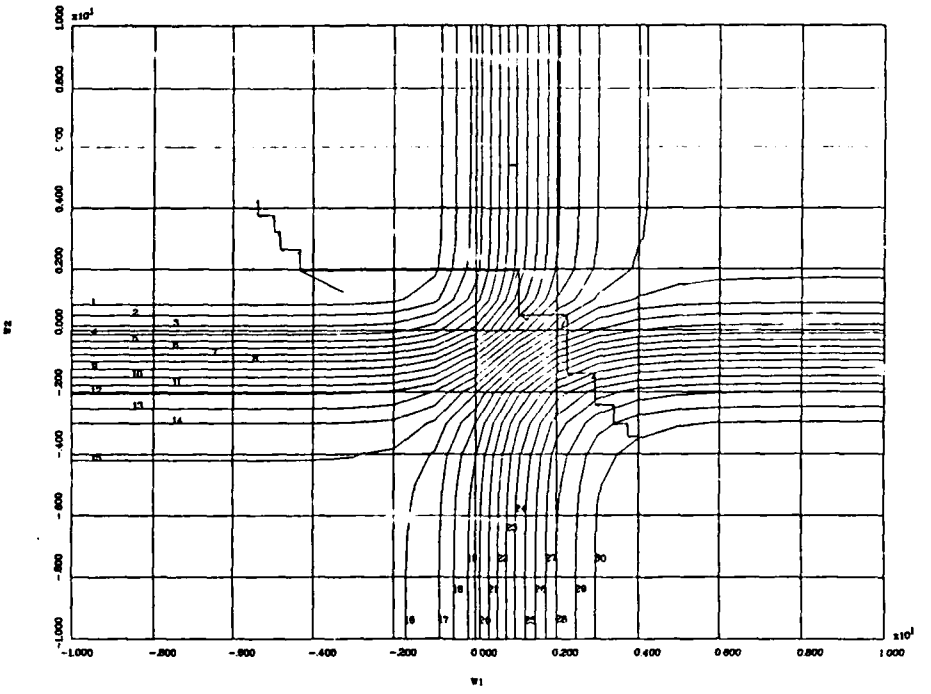


Figure 3.13 Input-by-Input Application of QSD

It simplifies the QSD algorithm by taking only a single step in the direction of the gradient at each iteration. Figure 3.14 illustrates its performance.

The algorithm of Figure 3.14 attains low error values as quickly as the full QSD algorithm (for this choice of starting point), but is unable to attain extremely small error surface values in a small number of steps. The plotting program used to generate Figure 3.14 does not show hundreds of very small jumps to the left of $W_1 = -4.0$. The algorithm terminated after the maximum allowed number of iterations, 240, at the point $(W_1, W_2) = (-4.82, 4.81)$.

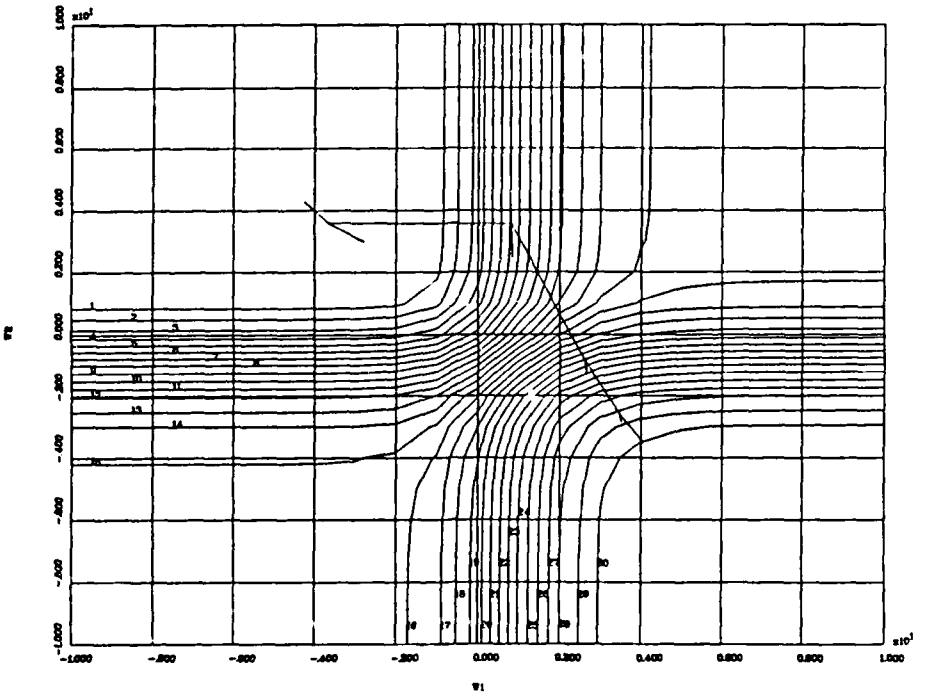


Figure 3.14 Single-Step Batch Processing

Figure 3.14 points out nicely an advantage of the QSD algorithm. The algorithm driving Figure 3.14 makes one step per update in the direction of the gradient, as in moving from P_0 to P_1 in Figure 3.9. But the size of the step is proportional to the slope of the surface (Equation (3.6)), so in regions of shallow slope, step sizes become minuscule and training requires very large numbers of updates. Hence the hundreds of tiny updates at the end. On the other hand, the QSD algorithm has the potential of moving directly to point P_M in Figure 3.9, regardless of the error surface gradient. Thus it attains a lower error value in far fewer steps (Figure 3.11).

Similar comparisons of the algorithms represented in Figures 3.11 through 3.14 for other choices of initial points (W_1, W_2) revealed that the QSD algorithm always attained very low error values most quickly. See Appendix D for sample plots from different initial points. Because its performance advantage comes at a low cost in programming complexity, a direct generalization of the QSD algorithm was chosen for implementation in the predictor network.

3.4.2 A Geometric Interpretation of Time Series Learning. Section 2.4 described how a three layer back propagation neural network can learn to "recognize" a plane region. The coordinates of points in the plane correspond to distinct network inputs. Whenever a point in the plane falls within the desired region, the net's output is near one; all other points in the plane yield network outputs near zero.

Time series prediction requires (unlike some pattern recognition networks) variable weights in the output neuron because network output values need to track time series values over a sometimes continuous range of numbers. Network outputs of only zero and one would suffice only in the case where the time series attained no more than two values. A hypothetical distribution of network weights will help to clarify the need for variable output node weights in a predictor network.

Consider a simple time series consisting of the sequence of values 0.5, 0.9, 0.5, 0.1, 0.5,... . Figure 3.15 shows that this sequence is actually a sampling at 90°

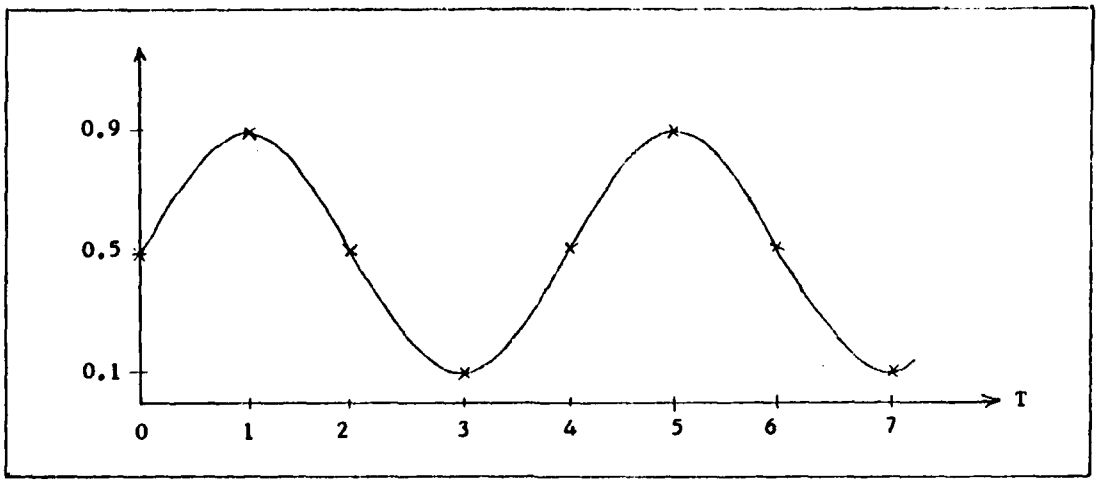


Figure 3.15 A Simple Time Series

intervals of a DC biased sine wave. Consider a predictor network with only two inputs, the two previous time samples. The network is assigned the task of learning through its weight update algorithm the following mapping of two dimensional input space into the range of values attained by the time series: (0.5,0.9) → 0.5; (0.9,0.5) → 0.1; (0.5,0.1) → 0.5; (0.1,0.5) → 0.9.

Figure 3.16 shows this mapping above the input plane (X_1, X_2). There are three regions in the plane corresponding to the three values attained by the time series. This suggests using three groups of first layer neurons to identify these regions. Figure 3.17 shows one way this can be done. Although there is complete interconnectivity of

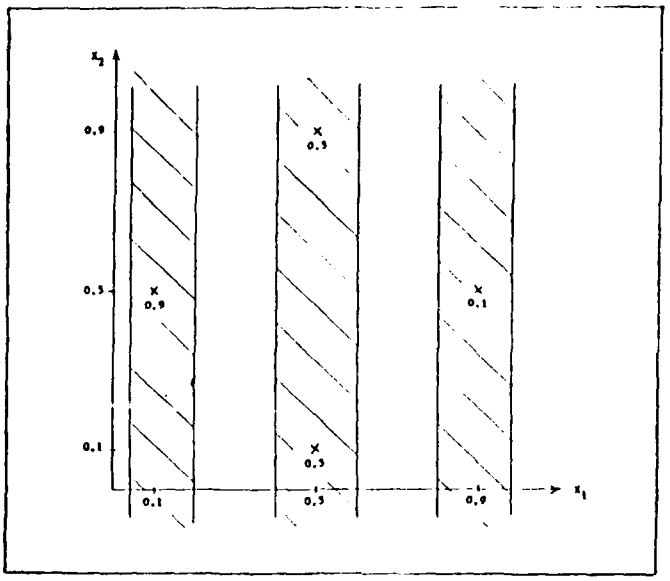


Figure 3.16 Mapping Sequence Pairs to Next Sequence Value

nodes between layers, for clarity only the connectivity of the nodes contributing to an output value of 0.9 is shown.

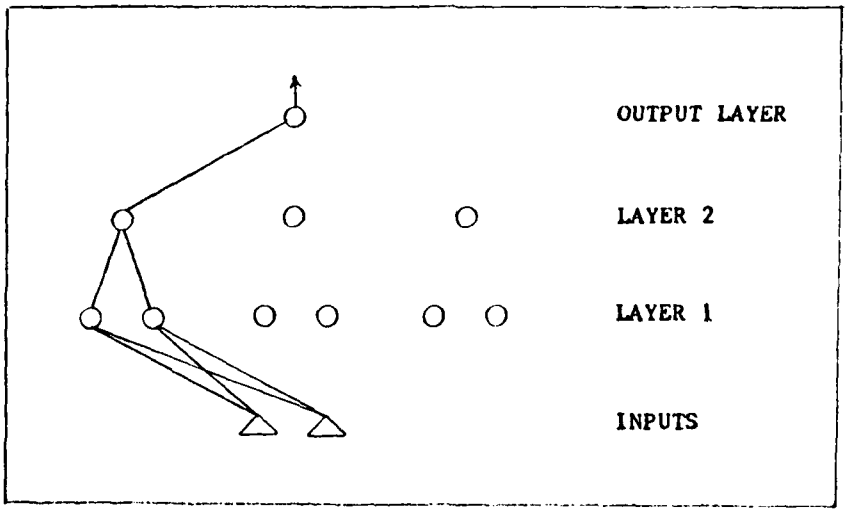


Figure 3.17 A Simple Predictor Network

Each of the layer one neurons corresponds to a vertical line in Figure 3.16. Since single neurons are capable of

distinguishing half-plane regions, proper selections of weights can enable two neurons to identify a region between two vertical lines [Lippmann87:14]. The two leftmost layer one neurons correspond to vertical lines which separate the point (0.1,0.5) from the other three input points. The remaining four neurons correspond to vertical lines which complete the demarcation in the input plane of three disjoint vertical bands containing the points 1) (0.1,0.5); 2) (0.5,0.1) and (0.5,0.9); and 3) (0.9,0.5).

Therefore, when an input pair falls in the leftmost band, the two leftmost layer one neurons both have output values near one. The leftmost layer two neuron works with these two neurons to produce a high output of its own in this situation. It does this by adjusting its weights to perform a logical AND operation on its associated two neurons [Lippmann87:16]. The weights connecting the leftmost layer two neuron to all other layer one neurons can be trained to values near zero, making the outputs of all other layer one neurons irrelevant.

The remaining two layer two neurons operate in similar fashion with the pairs of layer one neurons immediately below them. Thus when an input pair falls near (0.1,0.5), the output of the leftmost layer two neuron is near one and the outputs of the other layer two neurons are near zero. The output layer neuron therefore has an output Y given to a good approximation by $Y = 1/[1 + \exp(-WX + \theta)]$ (see Equation (2.3)), where the weight W and input X correspond to its

connection with the leftmost layer two neuron. The threshold θ can train to any fixed value. Since an input near (0.1,0.5) results in $X \approx 1$, the network output Y is therefore a monotonically increasing function of W (see Figure 2.4). Since W is trainable, it can assume a value which will give the desired output of 0.9 (if θ=0, then W=2.2).

Weights connecting the other layer two neurons to the output neuron can similarly be trained to yield Y values of 0.5 (when the middle neuron's output is high) and 0.1 (when the rightmost neuron's output is high).

Although the foregoing discussion has explained how a network could train its weights to solve a simple prediction problem, it does not imply that a network will in fact adjust its weights in this manner. Nor was the choice of plane regions in Figure 3.16 necessarily optimal in terms of neural net performance. Nevertheless, it provides a basis for a general rule for selecting the numbers of neurons needed on the first and second layers of a predictor network used for sine wave prediction. This rule is illuminated by consideration of a sine wave sampled at 45° intervals.

Figure 3.18 shows such a sine wave. Consider now a three input predictor network assigned the task of learning and then predicting values of this sine wave. This network must learn the following mapping of points in three-space:

$$A: \quad (0.5,0.78,0.9) \rightarrow 0.78$$

$$B: \quad (0.78,0.9,0.78) \rightarrow 0.5$$

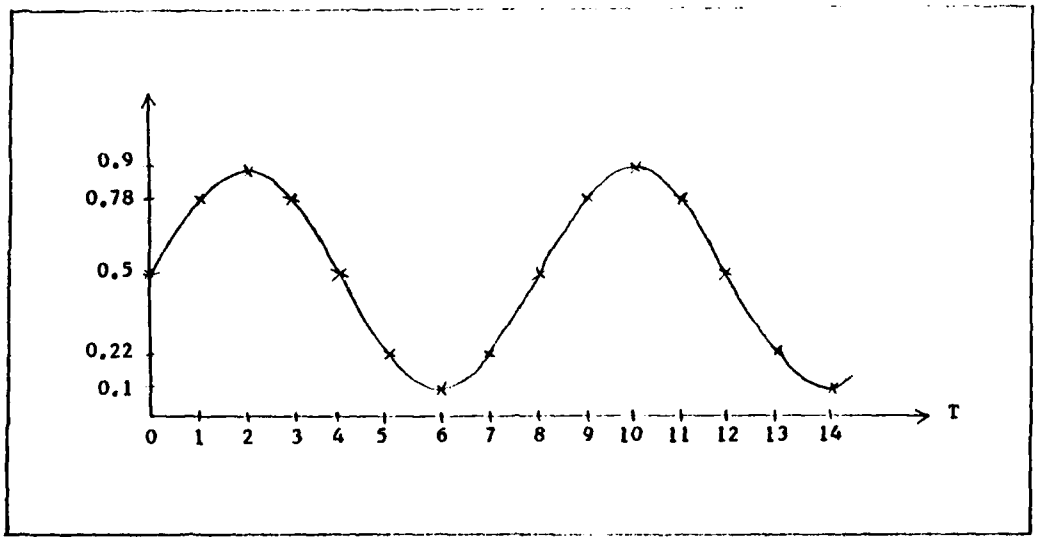


Figure 3.18 Another Time Series

C : (0.9,0.78,0.5) → 0.22

D : (0.78,0.5,0.22) → 0.1

E : (0.5,0.22,0.1) → 0.22

F : (0.22,0.1,0.22) → 0.5

G : (0.1,0.22,0.5) → 0.78

H : (0.22,0.5,0.78) → 0.9

In this case, there are five distinct output values; three of them occur twice, and two of them (the maximum and minimum values of the sine wave) occur only once.

This prediction process could be compared to a Gauss-Markov-3 model (the scalar output of a 3-state shaping filter). A Kalman filter could be built to do the desired prediction. The adaptive gain KF is then tantamount to adaptive weight selection by a neural network.

In two-space, distinct output values could be separated by parallel lines (Figure 3.16). In this example in three-space, all the pairs of points with equal output values can be separated by disjoint three-sided tubes. Two additional tubes can easily be selected to enclose the two remaining points in such a way that all five output values are represented by disjoint tubes.

To see this, consider projections of the points A through G in the space (X_1, X_2, X_3) onto the space (X_1, X_2) as shown in Figure 3.19. The lines A'G', B'F', and C'E' are

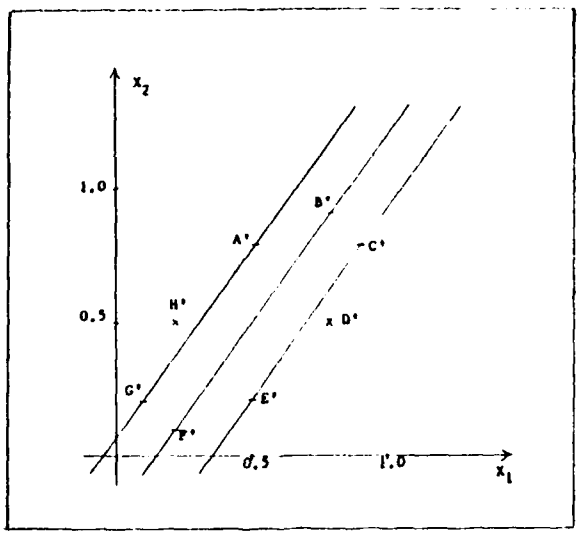


Figure 3.19 Parallel Projections in Two-space

mutually parallel, so the perpendicular planes containing the pairs of points (A,G), (B,F), and (C,E) are also parallel. Therefore tubes taken close to the lines AG, BF, and CE will provide disjoint separation, regardless of the slopes of the lines in three-space.

Just as a single neuron can distinguish half-plane regions in two-space by finding the line which separates them, it can (using three inputs instead of two) separate three-space into two regions by determining a plane [Lippmann87:13]. This suggests using three layer one neurons for each distinct output value, since each distinct output value is contained in a unique three-sided tube. One layer two neuron can then perform a logical AND operation on the three outputs to distinguish the inside of the tube (with an output near one) from the outside (with an output near zero). Figure 3.20 shows a network

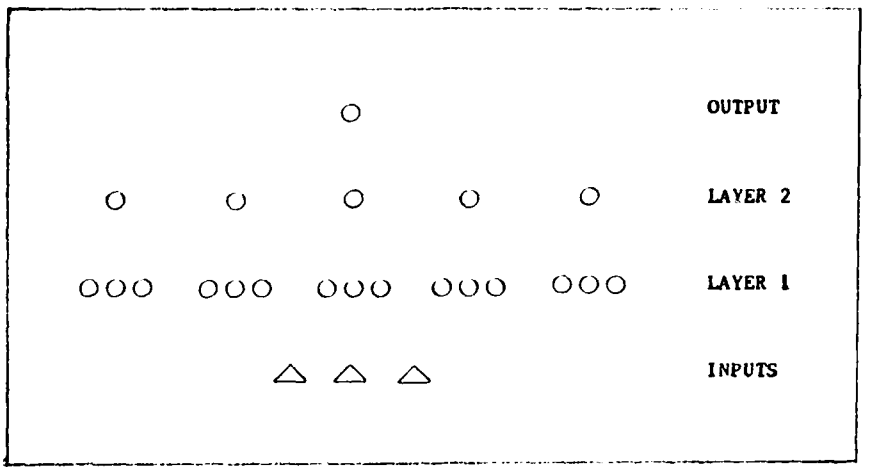


Figure 3.20 A Three Input Predictor Network

configuration which should therefore be adequate for predicting this time series. In fact, the neural network presented in Appendix C predicted the time series rather well using the configuration of Figure 3.20 (see Table 4.2).

Figures 3.17 and 3.20 suggest the following rule for determining numbers of nodes to be used when predicting sine waves. The number of first layer nodes can be chosen approximately equal to the product of the number of distinct values assumed by the times series and the number of network inputs. The number of second layer nodes can be chosen approximately equal to the number of distinct values assumed by the time series.

Predicting chaotic rather than periodic data introduces complexity to the partitioning of the input space. Chaotic data may be represented as tens of "cycles" of nonperiodic undulations appended to the sine wave of Figure 3.18 (see Figure 2.1). Initial values from the new data may fit adequately within existing tubes, but soon new data values (or old data values arising from greatly displaced input space coordinates) will force construction of new partitioning planes. Each new plane corresponds to a new layer one neuron. Furthermore, in instances where distant input vectors yield the same output, it may be necessary to partition equal data values disjointly, thereby adding to the number of second layer neurons required.

All of this suggests that large numbers of first and second layer neurons might be necessary for predicting chaotic data. But the geometric generalization from sine waves to chaotic data may be invalid. For example, weights may actually be updating in accord with the Fourier components of the underlying chaotic function. Furthermore,

the network's output is a continuous function of the network inputs, so some ability to interpolate between close learned output values might be expected. It turned out that both Lapedes' network [Lapedes88b:15] and the network of Appendix C were able to predict chaotic data using a total of only 20 hidden layer nodes. A parallel study being done at AFIT [Tarr88] is investigating optimization of the numbers of nodes for a given problem.

3.4.3 An Object-Oriented Design Predictor Network. It was recognized from the outset of this thesis effort that construction of a neural network which has the ability to predict the future (of a chaotic data stream) would raise far more questions about it than the author can begin to answer. Because of anticipated follow-on research, it was felt that the implementation of the network should be as understandable and modifiable as possible. Because the object-oriented design programming methodology (OOD) strongly supports these goals [Booch83:37], this methodology was chosen to design the network. The programming language selected was Ada, primarily because it directly supports the packaging construct essential to OOD.

Object-oriented design follows the following steps [Booch83:40]:

1. Define the problem
2. Develop an informal strategy for the problem domain
3. Formalize the strategy by
   a. Identifying the objects and their attributes

b. Identifying operations on the objects

c. Establishing the interfaces among the objects and operations, and

d. Deciding on implementations of the objects and operations.

The intention in this thesis is not to explain the network design in detail, but rather to provide to those familiar with OOD the all-important informal strategy used and a Booch diagram showing interfaces so that the source code provided in Appendix C will be more readable. It is hoped that the comments within the source code will allow quick comprehension of program detail.

The informal strategy may be stated in five sentences:

Initialize the network by obtaining essential network parameters. Assign initial values to all network weights. Get the time series values to be used in training and prediction. Train the network. Predict future time series values.

Objects were extracted from this informal strategy and used to form program packages. The "training" object was refined somewhat by extracting an "error surface" object. The Booch diagram showing the package visibilities is given in Figure 3.21.

The procedure JIMSPNET is the driving procedure of the program. Notice that the body of the procedure recapitulates the informal strategy.

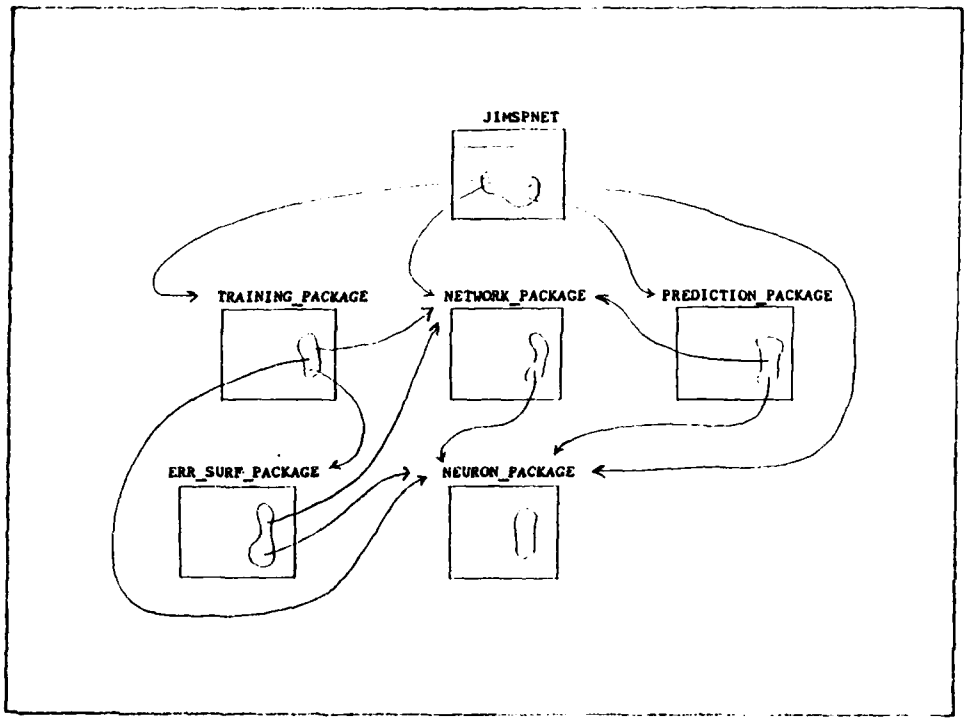


Figure 3.21 Booch Diagram of OOD Predictor Network

The NETWORK package contains the procedure which obtains from an input file named PARAMS five essential network parameters. These parameters are: the number of network inputs, the numbers of first and second layer neurons, the number of training vectors, and the number of values to be predicted. It also contains the procedure which obtains time series values from an input file named GMCDATA.

The TRAINING package is by far the largest package. It provides output to a file named TRNDATA to allow the program user to observe training performance.

The PREDICTION package includes the procedure PREDICT which provides predicted time series values to a file named

PRVALS. The values in PRVALS may then be compared to actual time series values.

3.5 Summary

This chapter has shown how chaotic data was generated. It compared the error surface performance of the QSD algorithm to some other weight update algorithms. A plausible explanation of weight updating in a predictor network was given, and the software architecture of a predictor network was outlined. The next chapter will present and discuss the results obtained using this network.

4. Results and Discussion

4.1 Introduction

The predictor network developed in the preceding chapter was tested first on data obtained by sampling sine waves at equally spaced intervals. When it was clear that the network was learning mappings of distinct input sequences to desired output values, the sine waves were sampled (after training) at intervals of the same length but at offset phases to observe how well the entire sine waves were being learned. Using the insight gained in these experiments, predictions were made using the mildly chaotic data obtained from the sum of incommensurate sine waves. Building on this experience, attempts were made to predict the more highly chaotic Glass-Mackey data.

4.2 Sine Wave Results

Figure 3.17 illustrates the first network configuration tested. This network was trained to predict a scaled version of the time series data shown in Figure 3.15. It was trained on the first full cycle of the sine wave, and instructed to predict time series values several cycles into the future at time steps $\Delta t = \pi/2$.

Because the chaotic data shown in Figure 3.3 falls between the values zero and two rather than zero and one, the network was designed to divide all input data by two and multiply all output data by two. This scaling allowed the

network output to express values larger than one, which is an upper bound of the sigmoidal output of every neuron.

Table 4.1 lists the results of this experiment. In

Table 4.1 Two Input Sine Wave Prediction

-T-	--X(T)--	-XPRED(T)-	-T-	--X(T)--	-XPRED(T)-
0	1.0	≡	13	1.8	1.79820
1	1.8	≡	14	1.0	1.00334
2	1.0	≡	15	0.2	0.203125
3	0.2	≡	16	1.0	0.993311
4	1.0	≡	17	1.8	1.79808
5	1.8	≡	18	1.0	1.01419
6	1.0	0.999954	19	0.2	0.203015
7	0.2	0.202788	20	1.0	0.972991
8	1.0	0.999666	21	1.8	1.79759
9	1.8	1.79831	22	1.0	1.05749
10	1.0	1.00063	23	0.2	0.202587
11	0.2	0.203136	24	1.0	0.892492
12	1.0	0.998394	25	1.8	1.79562

this table (and all subsequent tables in this chapter), T represents the number of Δt intervals to the right of zero on the t axis. The first predicted value XPRED(T) occurred at T = 6. It was the network output when the actual time series values X(4) = 1.0 and X(5) = 1.8 appeared at the inputs of the trained network. The second predicted value XPRED(7) was the network output when X(5) and XPRED(6) appeared at the inputs. The third and all later predicted values used predicted rather than actual time series values at both network inputs.

Figure 4.1 presents graphically the data of Table 4.1. Actual X(T) values occur at integer values of T, and are represented by asterisks; predicted values are circles. All remaining figures in this chapter use the same convention.

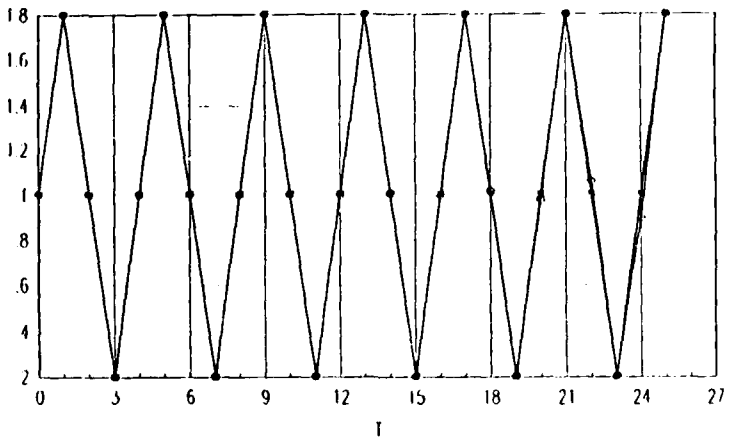


Figure 4.1 Two Input Sine Wave Prediction

Training the network with the first four input vectors required numerous adjustments of training parameters before the good results of Table 4.1 were obtained. Using notation similar to that found in the source code of Appendix C, the following parameter values were finally selected:

A1 = 0.5 (initial step size in the QSD algorithm)

MAXITERS = 800 (maximum number of times quadratic minima selected; a termination criterion)

WTSCLOSE = 10^{-20} (closest allowed distance between updated weights; checked at each iteration, this is also a termination criterion)

uwidth = 1.6 (distance from zero of endpoints of interval from which initial weight values are randomly chosen)

The extremely small value of WTSCLOSE was selected to ensure that MAXITERS would reach 800. Thus 800 linear descents of

the error surface were achieved, resulting in a very low error surface value. It was at this point that the prediction weights were fixed.

When the step size A1 was taken larger than about 0.5, the algorithm would sometimes terminate prematurely (in fewer than 800 iterations) because the minimizing point in weight space of the set of points (P_0, P_1, P_2) would be the initial point P_0 (see Appendix B). On the other hand, if A1 was taken larger than about 0.01, then 800 iterations were not enough to result in a low final error value.

When the initial weight interval uwidth was chosen less than about 0.5, the algorithm sometimes iterated on too small a subset of the weight space and resulted in a rather high final error value (coresponding to a high local minimum). When uwidth was chosen greater than about 5.0, sometimes all predicted values were very close or identical. In such cases, the sigmoidal outputs of some or all of the neurons could assume values very near their saturation values of zero or one.

The same parameters were used to predict values of the time series depicted in Figure 3.18, using the network configuration shown in Figure 3.20. Table 4.2 gives the results of this experiment, and Figure 4.2 shows the results graphically. Notice that the prediction accuracy has declined from the results of Table 4.1. This seems reasonable since the same number of updates are now taking place in a much higher dimensional weight space. The two

input network has 43 weights, and the three input network has 146. As the number of nodes in the network increases, it seemed reasonable that the number of update iterations should therefore be increased, too.

Table 4.2 Three Input Sine Wave Prediction

-T-	--X(T)--	-XPRED(T)-	-T-	--X(T)--	-XPRED(T)-
0	1.0000	≡	16	1.0000	1.0309
1	1.5657	≡	17	1.5657	1.5699
2	1.8000	≡	18	1.8000	1.7241
3	1.5657	≡	19	1.5657	1.5561
4	1.0000	≡	20	1.0000	0.9698
5	0.4343	≡	21	0.4343	0.4172
6	0.2000	≡	22	0.2000	0.2851
7	0.4343	≡	23	0.4343	0.4471
8	1.0000	≡	24	1.0000	1.0498
9	1.5657	≡	25	1.5657	1.5766
10	1.8000	≡	26	1.8000	1.7154
11	1.5657	1.6021	27	1.5657	1.5418
12	1.0000	1.0328	28	1.0000	0.9477
13	0.4343	0.4060	29	0.4343	0.4117
14	0.2000	0.2746	30	0.2000	0.2869
15	0.4343	0.4079			

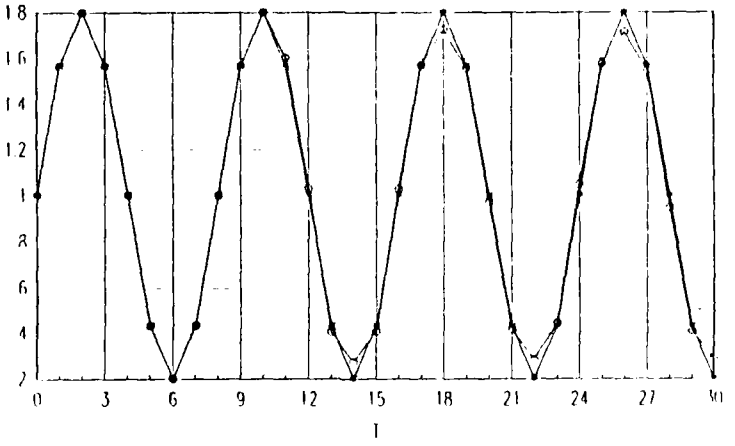


Figure 4.2 Three Input Sine Wave Prediction

One might ask whether the trained network which produced the results of Table 4.1 had come closer to "learning" the frequency of the sine wave, or to "learning" regions in the two dimensional input space in the vicinity of the inputs used for training (see Figure 3.16). In an attempt to answer this, the PREDICT procedure in Appendix C was modified slightly so that, following training, prediction could proceed immediately on data representing different inputs than any the network had yet seen. In the first case, after training with the input values 1.0, 1.8, 1.0, 0.2, and 1.0, the network was asked to provide an output when inputs of 0.95 and 0.15 were presented. Being near the learned input pair (1.0,0.2), the output might be expected to be slightly less than 1.0. In fact, however, the output was about 1.09. In the second case, after training, the network was asked to provide an output when inputs of 0.903 and 0.206 were presented. These inputs correspond to a 7° shift in the sine wave. Following the sine wave, an output somewhat greater than 1.0 (1.097, actually) might be expected. This was in fact the result, since an initial output of about 1.16 was obtained.

Although this experiment proved nothing, it did suggest that network learning may more accurately be described as an adaptation to the frequency of the input data than to its spatial orientation.

4.3 Incommensurate Sine Waves

The next experiments focused on predicting values of the data taken from a sum of sine waves with incommensurate frequencies. A short range of the function used is shown in Figure 3.6. Its fractal dimension was determined to be about 1.7 (Section 3.3). Applying Equation (2.4), the acceptable range of number of inputs was found to be from one to three, inclusive. For prediction purposes, samples were taken at intervals of $\Delta t = 0.8$. The network was configured most successfully with 3 inputs, 15 first layer nodes, and 5 second layer nodes. As in Section 4.2, some trial and error was required to find a good value for the parameter uwidth (in this case, 3.2). It was permitted to perform 1200 weight updates using the QSD algorithm before training was terminated and prediction initiated. Fewer updates resulted in less prediction accuracy, and more updates would probably have given greater accuracy. Performing 1200 updates required about 48 hours of VAX-11/780 batch time, however, and further updates would increase this time.

Table 4.3 shows the results of the experiment with the described parameter settings. Figure 4.3 presents the results graphically. Prediction began at $T = 28$, so training ended at the abscissa value $(T-1)\cdot\Delta t = 21.6$. Training was thus conducted over about four full cycles of each component sine wave. Extending the training interval

would probably have increased the prediction accuracy, but again at the expense of increased run time.

Table 4.3 Incommensurate Sines Prediction

-T-	--X(T)--	-XPRED(T)-	-T-	--X(T)--	-XPRED(T)-
.	.	≡	35	0.9830	0.7723
.	.	≡	36	0.8690	0.3059
.	.	≡	37	0.9955	0.8015
28	1.5751	1.5794	38	0.9237	1.5632
29	1.7976	1.7675	39	0.7746	1.5398
30	0.9562	0.8375	40	1.0272	0.7324
31	0.3257	0.2486	41	1.4661	0.3340
32	0.6767	0.7116	42	1.3067	0.8591
33	1.3146	1.5607	43	0.5543	1.5703
34	1.3532	1.6127	44	0.2881	1.4916

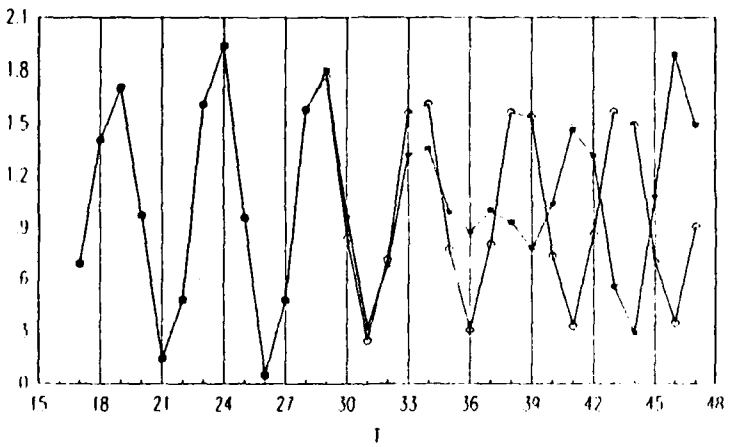


Figure 4.3 Incommensurate Sines Prediction

4.4 Glass-Mackey Prediction

The final data predicted was the Glass-Mackey data represented (over a short range) in Figure 2.1. In Section 3.3, the acceptable number of inputs was found to be two, three, or four. Training was initiated at time t = 60 and proceeded using samples taken at intervals $\Delta t = 6$ (following [Lapedes88a:6]). Best results were obtained using a network

configuration of 4 inputs, 10 layer one nodes, and 10 layer two nodes. Five hundred training vectors were used. Large numbers of training vectors demand long computer run times, so to keep the run time reasonable, only 40 weight updates were performed. Additional training would most likely have resulted in more accurate prediction.

The initial predicted values are presented in Table 4.4. Figure 4.4 shows these values graphically. Over 72 hours of VAX-11/780 batch time were needed to obtain these results.

Table 4.4 Glass-Mackey Prediction

-T-	--X(T)--	-XPRED(T)-	-T-	--X(T)--	-XPRED(T)-
.	.	≡	509	0.8472	0.8637
.	.	≡	510	0.6219	0.8045
.	.	≡	511	0.6231	0.8039
504	1.2290	1.1573	512	0.4902	0.8617
505	1.1303	1.1619	513	0.3704	0.9531
506	1.1242	1.1414	514	0.6344	1.0422
507	1.2658	1.0653	515	0.9531	1.1022
508	1.1946	0.9600	516	1.0803	1.1225

These predicted values fail to track the actual Glass-Mackey values after a much shorter period than any of the previous results. This can probably be attributed to the shortage of computer run time. The other predictions performed had sufficient run time to reduce the error function value by at least two orders of magnitude. The Glass-Mackey error value was reduced slightly less than one order of magnitude.

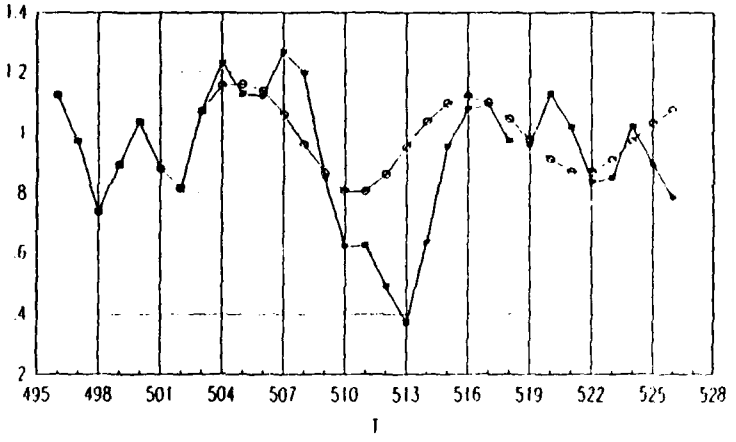


Figure 4.4 Glass-Mackey Prediction

4.5 Summary

This chapter has presented and discussed the prediction results obtained. The next chapter will give the conclusions and recommendations of this thesis.

5. Conclusions and Recommendations

5.1 Conclusions

This thesis has shown that a batch processing neural network which incorporates the efficient QSD training algorithm can predict chaotic data. In doing so, several insights into neural net prediction were gained.

Lapedes provided the rule $d < m+1 < 2d+1$ [Lapedes88a:6] for determining the number m of network inputs, given the fractal dimension d of the data being predicted. It was found that the largest values of m permitted by this rule gave the best prediction results.

Prediction accuracy can sometimes be improved by adjusting the parameter A1 in the QSD algorithm (given that all other network parameters remain unchanged). The A1 parameter is directly proportional to the initial step taken by the algorithm. Making it larger can improve training efficiency, although if it is taken too large the algorithm will terminate prematurely.

Accuracy can be improved in a number of other ways, but all come at the expense of increased computer run time. Prediction improves as:

1) the number of weight update iterations is increased. This has the effect of decreasing the value of the error function, which implies better "learning" of the training vectors.

2) the number of training vectors is increased. The network seems to gain a better feeling for the overall structure of the data as the number of samples which it "learns" is increased.

3) the number of layer one and two nodes is increased beyond a bare minimum. There may be a point beyond which increasing the number of hidden layer nodes is unproductive, but this was not investigated.

5.2 Recommendations

Prediction has been done for years using linear filtering techniques. Some results with these techniques would provide useful comparisons of prediction accuracy. Lapedes and Farber [Lapedes88a:7] claim greater accuracy with neural network prediction than, for example, Widrow-Hoff prediction.

The application of neural networks to the prediction of moving targets should be investigated. Problems where Markov-3 position processes are currently used are good candidates.

Equation (2.4), which is actually a pair of inequalities bounding the number m of network inputs needed by a predictor network, could be profitably investigated both empirically and theoretically. Perhaps the bounds could be fitted more tightly.

Computer run time was the most limiting factor in this research. The most obvious solution to this problem is to use a faster computer, but other alternatives are possible.

It was felt that the QSD algorithm would reduce run times considerably compared to a standard back-propagation network as described by Lippmann [Lippmann87:17]. It seems likely that it actually did. However, no attempt was made to quantify the advantages of the QSD algorithm in terms of run times. Many minimization algorithms are presented in numerical analysis texts and elsewhere (for example, [Maybeck88]). A comparative study of the run times required by different algorithms, perhaps using a simple standard prediction problem, would be quite valuable. The results may suggest a faster algorithm than any which have yet been applied to the prediction problem. A likely candidate might be a 2-stage search algorithm which begins by using the QSD algorithm and switches, when progress slows, to a Newton-Raphson technique.

A layer-by-layer, neuron-by-neuron explanation of how weights actually update to achieve prediction is lacking in the literature. As described in Chapter 3, a geometric interpretation of weight updating can provide good initial estimates of the numbers of hidden layer neurons required for prediction of sine wave data. The intuition gained from this interpretation does not extend well to chaotic data. It was found that surprisingly few neurons are required for chaotic prediction. An experimental result described in

Section 4.2 suggests that weights may actually update to correspond to frequency components of the input data. Further investigation along these lines would also be beneficial.

Finally, the program presented in Appendix C could no doubt be made more efficient. Care must be taken that efficiencies thus gained do not result in substantially reduced understandability or modifiability.

Appendix A: Glass-Mackey Computational Details

Introduction

Derivation details of Glass-Mackey generating algorithms were omitted in Chapter 3. This appendix supplies those details.

The Predictor-Corrector Method

Equation (3.1) is restated, then solved, giving Eq (3.2):

$$\frac{\Delta t}{2}\left\{ f(t + \Delta t) + f(t) \right\} \approx x(t + \Delta t) - x(t)$$

$$\frac{\Delta t}{2}\left\{ \frac{ax(t+\Delta t-\tau)}{1+x^{10}(t+\Delta t-\tau)} - bx(t+\Delta t) + \frac{ax(t-\tau)}{1+x^{10}(t-\tau)} - bx(t) \right\} \approx$$

$$x(t + \Delta t) - x(t)$$

$$x(t + \Delta t) \approx x(t)\left\{1 - \frac{b\Delta t}{2}\right\}$$

$$+ \left\{\frac{ax(t+\Delta t-\tau)}{1+x^{10}(t+\Delta t-\tau)} + \frac{ax(t-\tau)}{1+x^{10}(t-\tau)} - bx(t+\Delta t)\right\}$$

$$x(t+\Delta t)\left\{1+\frac{b\Delta t}{2}\right\} \approx x(t)\left\{\frac{2-b\Delta t}{2}\right\}$$

$$+ \left\{\frac{ax(t+\Delta t-\tau)}{1+x^{10}(t+\Delta t-\tau)} + \frac{ax(t-\tau)}{1+x^{10}(t-\tau)}\right\}\frac{\Delta t}{2}$$

$$x(t+\Delta t) \approx \frac{2-b\Delta t}{2+b\Delta t}\, x(t) + \frac{\Delta t}{2+b\Delta t}\left\{\frac{ax(t-\tau)}{1+x^{10}(t-\tau)} + \frac{ax(t+\Delta t-\tau)}{1+x^{10}(t+\Delta t-\tau)}\right\}$$

as desired.

The Integrating Factor Method

Equation (3.3) is here derived, beginning with Equation (2.1):

$$\frac{dx(t)}{dt} = \frac{ax(t-\tau)}{1 + x^{10}(t-\tau)} - bx(t)$$

$$\exp(bt)\,\frac{dx(t)}{dt} + b\exp(bt)\,x(t) = \frac{ax(t-\tau)}{1 + x^{10}(t-\tau)}\exp(bt)$$

Letting $t = \tau + k\Delta t$ (where k is a positive integer) and s be a dummy variable of integration replacing t,

$$\frac{d[\exp(bs)x(s)]}{ds} = \frac{ax(s-\tau)}{1 + x^{10}(s-\tau)}\exp(bs)$$

$$\int_{\tau}^{t} \frac{d[\exp(bs)x(s)]}{ds}\, ds = \int_{\tau}^{t} \frac{ax(s-\tau)}{1 + x^{10}(s-\tau)}\exp(bs)\, ds$$

$$\exp(bt)\,x(t) - \exp(b\tau)\,x(\tau) = \int_{\tau}^{t} \frac{ax(s-\tau)}{1 + x^{10}(s-\tau)}\exp(bs)\, ds$$

Letting $G(s) = \int_{\tau}^{t} \frac{ax(s-\tau)}{1 + x^{10}(s-\tau)}\exp(bs)\, ds$,

$$x(t) = \exp[b(\tau-t)]\, x(\tau) + \exp(-bt)\, G(s)$$

$$= \exp(-bk\Delta t)\, x(\tau) + \exp(-bt)\, G(s)$$

Dividing the interval $[\tau,t]$ into adjacent intervals of width Δt, to a good approximation the area represented by $G(s)$ is a sum of areas of narrow trapezoids:

$$G(s) = \frac{\Delta t}{2}\left\{ G(\tau)+G(\tau+\Delta t) \right\} + \frac{\Delta t}{2}\left\{ G(\tau+\Delta t)+G(\tau+2\Delta t) \right\}$$

$$+ \cdots + \frac{\Delta t}{2}\left\{ G[\tau+(k-1)\Delta t] + G(\tau+k\Delta t) \right\}$$

$$= \Delta t \left\{ \frac{G(\tau)}{2} + G(\tau+\Delta t) + \cdots + G[\tau+(k-1)\Delta t] + \frac{G(\tau+k\Delta t)}{2} \right\}$$

Thus

$$x(t) = \exp(-bk\Delta t)\, x(\tau) + \Delta t \exp(-bt)$$

$$\cdot\left\{ \frac{G(\tau)}{2} + G(\tau+\Delta t) + \cdots + G[\tau+(k-1)\Delta t] + \frac{G(\tau+k\Delta t)}{2} \right\}$$

$$x(\tau+k\Delta t) = \exp(-bk\Delta t)\, x(\tau) + \Delta t \exp[-b(\tau+k\Delta t)]$$

$$\cdot\left\{ \frac{G(\tau)}{2} + G(\tau+\Delta t) + \cdots + G[\tau+(k-1)\Delta t] + \frac{G(\tau+k\Delta t)}{2} \right\}$$

This final equation is (3.3), as desired.

Appendix B: The QSD Algorithm

As mentioned in Section 3.4.1, Dahl suggested using a quadratic approximation to an error surface cross section as a means of reducing the number of steps required to attain convergence of weights to a minimum value [Dahl87:529]. The "Q" in the name QSD Algorithm gives credit to Dr. Dennis Quinn of AFIT's department of mathematics who outlined the details of one such approach to the student "S", who here provides a detailed derivation.

For clarity, a two-dimensional weight space is assumed, but the generalization to weight spaces of arbitrary dimension is trivial. A generalization is implemented in the program source code of Appendix C in the body of the network training package.

Figure B.1 shows a contour representation of weight

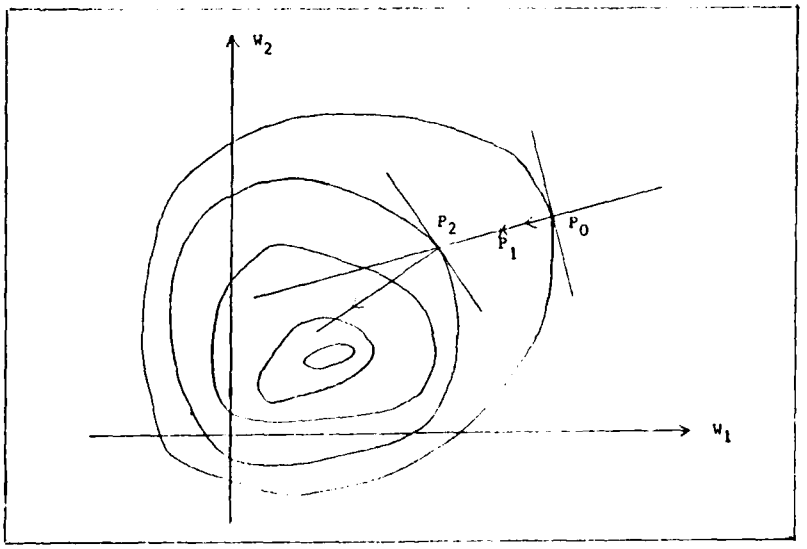


Figure B.1 Error Surface Contours

space. The minimizing procedure starts with a point P_0 = $(W_1(0),W_2(0))$ at time zero. This point is chosen randomly but in such a way that both components are small enough to avoid sigmoidal saturation when applied to a neuron. Sigmoidal saturation occurs when a neuron's output value is very near either zero or one; it can lead to numerical difficulties and poor network performance.

The general approach is to evaluate E and ∇E at P_0 and, moving in the direction of ∇E according to a quadratic approximation of E, find two additional potentially minimizing points P_1 and P_2. The point P_1 corresponding to the minimum of the set $\{E(P_0),E(P_1),E(P_2)\}$ becomes the next point P_0 for which a new gradient is found and from which the algorithm continues in its search for an error surface minimum. The algorithm stops when consecutive initial points P_n and P_{n+1} are within some small predetermined distance of each other in weight space.

The gradient $\nabla E(P_0)$ is a vector which determines a plane perpendicular to the (W_1,W_2) plane. A cross section of $E(W_1,W_2)$ in this perpendicular plane is labeled E(A) in Figure B.2. The points P_0, P_1, P_2, and P_M are here understood to be points in the (W_1,W_2) plane which lie on the A axis.

Points Q on the A axis admit two equivalent representations. A point Q may be represented by a scalar ψ where

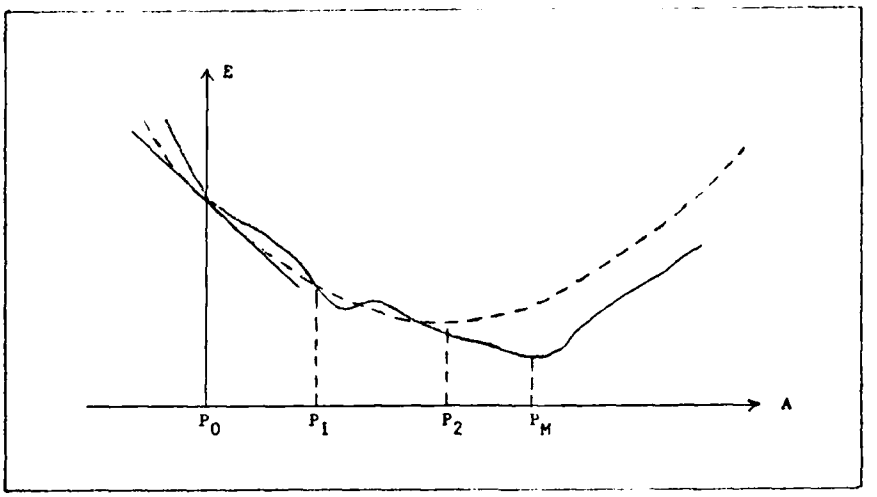


Figure B.2 When a Quadratic Approximation Minimizes E

$$Q = P_0 - \psi \frac{dE}{dA} (P_0)$$

and Q itself is a scalar corresponding to a location on the A axis. Alternately, Q may be considered a point in (W_1, W_2) space given by a scalar ζ and the vector equation

$$\underline{Q} = \underline{P}_0 - \zeta \Delta E(\underline{P}_0)$$

In both representations, Q approaches P_0 as ψ (or ζ) approaches zero. The latter representation is of practical importance in updating weights, but sometimes the former is more enlightening. In general, the P_i are understood to be points in (W_1, W_2) space and the vector notation is omitted.

As illustrated in Figure 3.8, for small positive numbers ψ, $E(Q) < E(P_0)$. Given P_0, therefore, a small positive number A_1 is chosen and the point P_1 is determined by $P_1 = P_0 - A_1 \nabla E(P_0)$.

In practice, the small positive number A_1 corresponds to ζ in the vector equation $\underline{Q} = \underline{P}_0 - \zeta \nabla E(\underline{P}_0)$. It is found by trial and error, based on network performance, and values between 1.5 and 3 seemed to work best with most chaotic time series data.

It is best to pick A_1 smaller than essential, as illustrated in Figure B.2, to ensure that $E(P_1) < E(P_0)$ in most cases. The values $E(P_1) \triangleq E_1$ and $E(P_0) \triangleq E_0$ are computed on the basis of actual network outputs as indicated in Equation (3.5) and are true error function values. But using these values, it may be possible to estimate the location of a point P_M on the A axis which minimizes the value of E in the cross section. To this end, approximate E as a quadratic function of the variable A by the equation $E = \alpha + \beta A + \gamma A^2$. The parameters α, β, and γ can be determined, the derivative of E set to zero, and the value of A which solves this equation will correspond to an extremum (hopefully a minimum) of the quadratic approximation to E. This value of a corresponds to the point labeled P_2 in Figure B.2.

Points P along the A axis are given by $P = P_0 - A \nabla E(P_0)$. This relationship defines a metric on A and allows the explicit representation $E(P) = E[P_0 - A \nabla E(P_0)]$. The function E may be considered a function of $A(W_1, W_2)$; the variable A is actually a function of W_1 and W_2. Conversely, W_1 and W_2 may be considered functions of A, since $P = P_0 - A \nabla E(P_0)$ may be written as the pair of equations

$$W_1(P) = W_1(A) = W_1(P_0) - A \cdot \frac{\partial E}{\partial W_1}(P_0)$$

$$W_2(P) = W_2(A) = W_2(P_0) - A \cdot \frac{\partial E}{\partial W_2}(P_0)$$

where $W_i(P)$ denotes the ith weight component of P.

It is thus meaningful to write $E = E(P) = E(A) = E[A(W_1, W_2)] = E[W_1(A), W_2(A)] = \alpha + \beta A + \gamma A^2$. Choosing P_0 to lie at the origin of the A axis, α may be determined simply by $E(P_0) = E[P_0 - 0 \nabla E(P_0)] = E(0) = \alpha = E_0$. Finding β and γ will require an examination of the derivative of E with respect to A.

Invoking the chain rule,

$$\frac{dE}{dA} = \frac{\partial E}{\partial W_1}\frac{dW_1}{dA} + \frac{\partial E}{\partial W_2}\frac{dW_2}{dA} = \beta + 2\gamma A$$

But $\frac{dW_1}{dA}$ is simply the constant $-\frac{\partial E}{\partial W_1}(P_0)$, and $\frac{dW_2}{dA}$ is the constant $-\frac{\partial E}{\partial W_2}(P_0)$. Evaluating $\frac{dE}{dA}$ at P_0 (that is, at A = 0),

$$-\left\{\frac{\partial E}{\partial W_1}(P_0)\right\}^2 - \left\{\frac{\partial E}{\partial W_2}(P_0)\right\}^2 = \beta$$

which can be written more simply as $\beta = -|\nabla E(P_0)|^2$.

Thus $E(A) = \alpha + \beta A + \gamma A^2 = E_0 - |\nabla E(P_0)|^2 A + \gamma A^2$. This may be solved at $A = A_1$ to obtain

$$\gamma = \frac{E_1 - E_0 + |\nabla E(P_0)|^2 A_1}{A_1^2}$$

All the parameters of the quadratic approximation have now been obtained. Setting the derivative of E(A) to zero and solving for the variable A now gives the parameter A_2 of the potential minimizing point P_2:

$$A = \frac{A_1^2 |\nabla E(P_0)|^2}{2 [E_1 - E_0 + |\nabla E(P_0)|^2 A_1]^2} \triangleq A_2$$

This point P_2 is given by $P_2 = P_0 - A_2 \nabla E(P_0)$. If E is concave downward near P_0, this may represent a maximizing rather than a minimizing value. Figure B.3 illustrates this

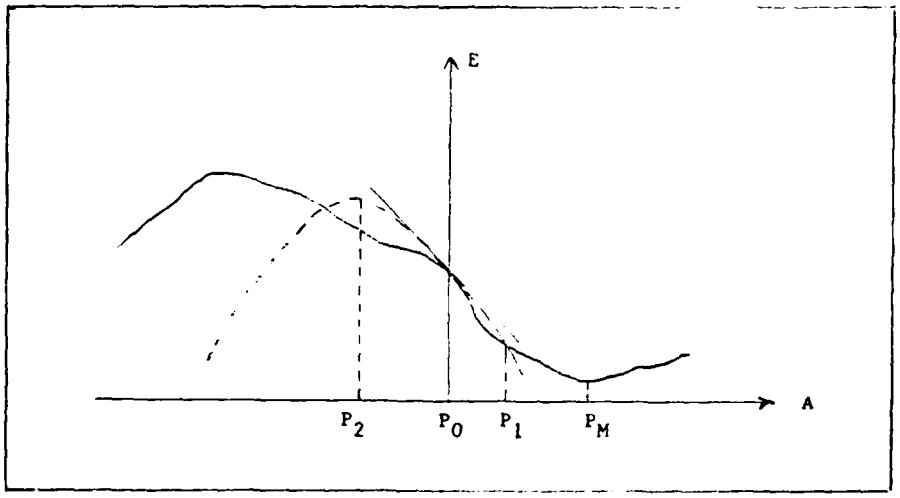


Figure B.3 When a Quadratic Approximation Maximizes E

situation. The QSD algorithm thus takes the point P_1 corresponding to the minimum of the set of values $\{E(P_0), E(P_1), E(P_2)\}$ as the next initial point P_0. It terminates when any two consecutive starting points P_n and

P_{n+1} are within a predetermined small distance of each other.

Appendix C: Ada Source Code

This appendix contains the Ada code which implements the predictor network developed in this thesis. The code was hosted on a VAX 11-780 computer located in the Information Sciences Laboratory at AFIT.

The Ada packages are presented in order of visibility, beginning with the main procedure, JIMSPNET. The order of visibility may be determined from Figure 3.20. Package specifications are always presented immediately before the corresponding package bodies.

In experimenting with new data streams, it is typically necessary to adjust several network parameters to obtain good results. The package TRAINING_PACKAGE contains three such parameters. Parameter A1 is the initial step size used by the QSD algorithm. As the number of network nodes increases, A1 should generally be reduced. Parameter MAX_ITERS is the upper bound on the number of weight updates performed. More accurate predictions usually result as MAX_ITERS is increased (the error surface value is driven lower), but large MAX_ITERS values require a great deal of computer run time. The parameter WTS_CLOSE is a termination criterion; when updated weight coordinates are within Euclidean distance WTS_CLOSE of the weight coordinates immediately preceding them, training ceases. For experimentation, WTS_CLOSE was usually assigned a very small

value so that training would terminate only after MAX_ITERS updates.

Package NETWORK_PACKAGE contains the parameter u_width which determines how close the initial random weights are to zero. Usually u_width values near 1.6 gave good results. Very large values of u_width (ten or above) often cause the network output to quickly stabilize at a constant value, regardless of variations in input data.

```
with NEURON_PACKAGE;                    use NEURON_PACKAGE;
with NETWORK_PACKAGE;                   use NETWORK_PACKAGE;
with TRAINING_PACKAGE;                  use TRAINING_PACKAGE;
with PREDICTION_PACKAGE;                use PREDICTION_PACKAGE;

procedure JIMSPNET is
   --This is the highest level controller procedure of the
   --predictor network.  It first obtains from an input file
   --named PARAMS the numbers N0,N1, and N2 of neurons in
   --the input, first, and second layers respectively.  It
   --obtains from the same file the number NBR_TVS of
   --training vectors to be used and the number NBR_PVS of
   --time series values to be predicted.  Procedure
   --GET_PARAMS does all of this.  The sizes of the various
   --network arrays are then declared, as well as the size
   --of the one-dimensional arrays X and X_PRED which hold
   --actual and predicted time series values.

   --It was written by Jim Stright at AFIT in August 1988.

   N0,N1,N2,NBR_TVS,NBR_PVS :positive;
   MAX2,MAX3                :positive;
   DESIRED_OUT              :float; --last trng vector comp.
   VALUE                    :float; --value of partial der.
                                    --of error function
   GRAD_SQ                  :float; --square of error fctn
                                    --gradient
   ERROR                    :float; --value of error fctn at
                                    --a point in wt. space
begin
   GET_PARAMS(N0,N1,N2,NBR_TVS,NBR_PVS);

   --set MAX2 = max(N1,N2)
   if N1 > N2 then
      MAX2:= N1;
   else
      MAX2:= N2;
   end if;

   --set MAX3 = max(N0,N1,N2)
   if N0 > MAX2 then
      MAX3:= N0;
   else
      MAX3:= MAX2;
   end if;

   declare
      --The following three array types are indexed to
      --specify a particular node in the network.  The
      --first element of each array denotes the layer
      --number in which the node occurs.  The second
      --element is the number of the node within that layer
      --(the nodes are numbered from left to right).  If
```

```
      --the third element is zero (in the case of
      --WEIGHT_TYPE), the weight is a threshold.  Otherwise
      --the third element denotes a connection to the node
      --or input with this number in the layer below.

      W,W1,W2,WH:WEIGHT_TYPE(1..3,1..MAX2,0..MAX3);
      I: INPUT_TYPE(1..3,1..MAX2,1..MAX3);
      O: OUTPUT_TYPE(1..3,1..MAX2);

      --The network has N0 inputs:
      NET_IN: NET_INPUT_TYPE(1..N0);

      --It is necessary to supply a total of
      --N0+NBR_TVS+NBR_PVS time series values for
      --comparison to predicted values X_PRED.
      X,X_PRED: TS_VALUES(0..N0+NBR_TVS+NBR_PVS-1);

   --Procedure SET_INIT_WTS assigns initial values to all of
   --the weights in the network.  GET_TS_VALUES loads the
   --the one dimensional array X sequentially with points of
   --the time series taken from an input file GMCDATA.
   --Starting at the initial weights, TRAIN finds a point in
   --weight space which minimizes the value of the error
   --function determined by the training vectors (which are

   --extracted from the X array).  An output file TRNDATA
   --saves certain values obtained in the training process
   --for possible use in tuning the network.  Using the
   --final weights from TRAIN, PREDICT sends network output
   --values sequentially to the array X_PRED and lists them
   --in an output file named PRVALS.

   begin
      SET_INIT_WTS(N0,N1,N2,W);
      GET_TS_VALUES(N0,NBR_TVS,NBR_PVS,X,X_PRED);
      TRAIN(N0,N1,N2,NBR_TVS,X,W,W1,W2,WH,NET_IN,I,O,
            DESIRED_OUT,VALUE,GRAD_SQ,ERROR);
      PREDICT(N0,N1,N2,NBR_TVS,NBR_PVS,X,W,X_PRED,
              NET_IN,O,I);
   end; --of declare block

end JIMSPNET;
```

```
with NETWORK_PACKAGE;        use NETWORK_PACKAGE;
with NEURON_PACKAGE;         use NEURON_PACKAGE;

package PREDICTION_PACKAGE is

   procedure GET_PRED_VECTOR(P_CT,N0,NBR_TVS: in positive;
                             X_PRED: in TS_VALUES;
                             NET_IN: out NET_INPUT_TYPE);
   procedure PREDICT(N0,N1,N2,NBR_TVS,NBR_PVS: in positive;

               X: in TS_VALUES;
                  W: in WEIGHT_TYPE;
                  X_PRED: in out TS_VALUES;
                  NET_IN: in out NET_INPUT_TYPE;
                  O: in out OUTPUT_TYPE;
                  I: in out INPUT_TYPE);

end PREDICTION_PACKAGE;
```

```
with system;
with text_io;              use text_io;
with float_text_io;        use float_text_io;
with integer_text_io;      use integer_text_io;

package body PREDICTION_PACKAGE is

   procedure GET_PRED_VECTOR(P_CT,N0,NBR_TVS: in positive;
                             X_PRED: in TS_VALUES;
                             NET_IN: out NET_INPUT_TYPE) is
   --Gets prediction vector number P_CT and enters it in
   --the NET_IN array.

   begin
      for C1 in 1..N0 loop
         NET_IN(C1):= X_PRED(NBR_TVS + P_CT + C1 - 2)/2.0;
         --Division by 2 keeps network inputs in the range
         --[0,1] since all time series values are in the
         --range [0,2].

      end loop;
   end GET_PRED_VECTOR;

   procedure PREDICT(N0,N1,N2,NBR_TVS,NBR_PVS: in positive;
                     X: in TS_VALUES;
                     W: in WEIGHT_TYPE;
                     X_PRED: in out TS_VALUES;
                     NET_IN: in out NET_INPUT_TYPE;
                     O: in out OUTPUT_TYPE;
                     I: in out INPUT_TYPE) is

   --Makes table in PRVALS of actual versus predicted values
   --based on trained weights in the W array.  The first
   --prediction vector which GET_PRED_VECTOR assigns to the
   --NET_IN array has time series point number NBR_TVS as
   --its first component and time series point number
   --NBR_TVS+N0-1 as its last component.  Procedure
   --COMPUTE_NETWORK_OUTPUT then provides the first
   --predicted value, which occurs at time series point
   --number NBR_TVS+N0.  The loop steps this process across
   --NBR_PVS training vectors.

      OUTFILE: text_io.file_type;
   begin
      create(OUTFILE,out_file,"PRVALS");
      put(OUTFILE,"       --T--"); set_col(OUTFILE,15);
      put(OUTFILE,"   --X(T)--"); set_col(OUTFILE,30);
      put(OUTFILE," --X_PRED(T)--");
      new_line(OUTFILE);
      for C1 in 1..NBR_PVS loop
         GET_PRED_VECTOR(C1,N0,NBR_TVS,X_PRED,NET_IN);
         COMPUTE_NETWORK_OUTPUT(N0,N1,N2,NET_IN,W,I,O);
         X_PRED(N0 + NBR_TVS - 1 + C1):= O(3,1)*2.0;
```

```
            --Multiplication by 2 restores time series values
            --(which had been divided by 2 at the network
            --inputs) to their original range of [0,2].

            put(OUTFILE,N0 + NBR_TVS - 1 + C1);
            set_col(OUTFILE,15);
            put(OUTFILE,X(N0 + NBR_TVS - 1 + C1));
            set_col(OUTFILE,30);
            put(OUTFILE,X_PRED(N0+ NBR_TVS - 1 + C1));
            new_line(OUTFILE);
         end loop;
         close(OUTFILE);
      end PREDICT;

end PREDICTION_PACKAGE;
```

```
with NEURON_PACKAGE;          use NEURON_PACKAGE;
with NETWORK_PACKAGE;         use NETWORK_PACKAGE;
with ERR_SURF_PACKAGE;        use ERR_SURF_PACKAGE;

package TRAINING_PACKAGE is
   procedure PART_DER_LYR3(NODE_BEL: in natural;
                           N0,N1,N2,NBR_TVS: in positive;
                           X: in TS_VALUES;
                           NET_IN: in out NET_INPUT_TYPE;
                           W: in WEIGHT_TYPE;
                           I: in out INPUT_TYPE;
                           O: in out OUTPUT_TYPE;
                           DESIRED_OUT: in out float;
                           VALUE: out float);

   procedure PART_DER_LYR2(NODE: in positive;
                           NODE_BEL: in natural;
                           N0,N1,N2,NBR_TVS: in positive;
                           X: in TS_VALUES;
                           NET_IN: in out NET_INPUT_TYPE;
                           W: in WEIGHT_TYPE;
                           I: in out INPUT_TYPE;
                           O: in out OUTPUT_TYPE;
                           DESIRED_OUT: in out float;
                           VALUE: out float);

   procedure PART_DER_LYR1(NODE: in positive;
                           NODE_BEL: in natural;
                           N0,N1,N2,NBR_TVS: in positive;
                           X: in TS_VALUES;
                           NET_IN: in out NET_INPUT_TYPE;
                           W: in WEIGHT_TYPE;
                           I: in out INPUT_TYPE;
                           O: in out OUTPUT_TYPE;
                           DESIRED_OUT: in out float;
                           VALUE: out float);

   procedure FIND_P1(N0,N1,N2,NBR_TVS: in positive;
                     X: in TS_VALUES;
                     NET_IN: in out NET_INPUT_TYPE;
                     W,W1: in out WEIGHT_TYPE;
                     I: in out INPUT_TYPE;
                     O: in out OUTPUT_TYPE;
                     DESIRED_OUT,VALUE: in out float);

   procedure FIND_P2(N0,N1,N2,NBR_TVS: in positive;
                     X: in TS_VALUES;
                     NET_IN: in out NET_INPUT_TYPE;
                     W,W2: in out WEIGHT_TYPE;
                     I: in out INPUT_TYPE;
                     O: in out OUTPUT_TYPE;
                     DESIRED_OUT,VALUE: in out float);
```

```
    procedure MOVE_WT_VALS(N0,N1,N2: in positive;
                           WTS1: in WEIGHT_TYPE;
                           WTS2: out WEIGHT_TYPE);

    procedure COMPUTE_GRAD_SQ(N0,N1,N2,NBR_TVS: in positive;
                              X: in TS_VALUES;
                              NET_IN: in out NET_INPUT_TYPE;
                              W: in WEIGHT_TYPE;
                              I: in out INPUT_TYPE;
                              O: in out OUTPUT_TYPE;
                              DESIRED_OUT,VALUE:in out float;
                              GRAD_SQ: out float);

    function DIST_TO_PT(N0,N1,N2: in positive;
                        W,WTS: in WEIGHT_TYPE) return float;

    procedure TRAIN(N0,N1,N2,NBR_TVS: in positive;
                    X: in TS_VALUES;
                    W,W1,W2,WH: in out WEIGHT_TYPE;
                    NET_IN: in out NET_INPUT_TYPE;
                    I: in out INPUT_TYPE;
                    O: in out OUTPUT_TYPE;
                    DESIRED_OUT,VALUE,GRAD_SQ,ERROR: in out
                                                     float);

end TRAINING_PACKAGE;
```

```
with system;
with float_math_lib;       use float_math_lib;
with math_lib_extension;   use math_lib_extension;
with text_io;              use text_io;
with float_text_io;        use float_text_io;
with integer_text_io;      use integer_text_io;

package body TRAINING_PACKAGE is

   A1         :float:=0.1;      --step size to point P1
   A2         :float;           --step size to point P2
   A2D        :float;           --denominator of A2
   E0         :float;           --error value at P0
   E0_1ST     :float;           --initial error value at P0
   E1         :float;           --step #1 error value
   E2         :float;           --step #2 error value
   MAX_ITERS  :natural:=400;    --max nbr of weight updates
   IT_CT      :natural:=0;      --counts nbr of weight updates
   WTS_CLOSE  :float:=1.0E-20; --termination criterion

   procedure PART_DER_LYR3(NODE_BEL: in natural;
                           N0,N1,N2,NBR_TVS: in positive;
                           X: in TS_VALUES;
                           NET_IN: in out NET_INPUT_TYPE;
                           W: in WEIGHT_TYPE;
                           I: in out INPUT_TYPE;
                           O: in out OUTPUT_TYPE;
                           DESIRED_OUT: in out float;
                           VALUE: out float) is
   --Finds VALUE of partial derivative of error function wrt
   --a particular layer 3 weight W(3,1,NODE_BEL)
   --at the point W in weight space.

      SUM     :float:=0.0;
   begin
      for I1 in 1..NBR_TVS loop
         GET_TRNG_VECTOR(I1,N0,X,NET_IN,DESIRED_OUT);
         COMPUTE_NETWORK_OUTPUT(N0,N1,N2,NET_IN,W,I,O);
         --Also computes intermediate outputs of neurons
         --throughout the network.

         if NODE_BEL = 0 then
            SUM:= SUM - (O(3,1)-DESIRED_OUT)*O(3,1)
                  *(1.0 - O(3,1));
         else
            SUM:= SUM + (O(3,1)-DESIRED_OUT)*O(3,1)
                  *(1.0-O(3,1))*I(3,1,NODE_BEL);
         end if;
      end loop;
      VALUE:= 2.0 * SUM;
   end PART_DER_LYR3;

   procedure PART_DER_LYR2(NODE: in positive;
```

```
                                NODE_BEL: in natural;
                                N0,N1,N2,NBR_TVS: in positive;
                                X: in TS_VALUES;
                                NET_IN: in out NET_INPUT_TYPE;
                                W: in WEIGHT_TYPE;
                                I: in out INPUT_TYPE;
                                O: in out OUTPUT_TYPE;
                                DESIRED_OUT: in out float;
                                VALUE: out float) is
--Finds VALUE of partial derivative of error function wrt
--a particular layer 2 weight W(2,NODE,NODE_BEL)
--at the point W in weight space.

   SUM    :float:=0.0;
begin
   for I1 in 1..NBR_TVS loop
      GET_TRNG_VECTOR(I1,N0,X,NET_IN,DESIRED_OUT);
      COMPUTE_NETWORK_OUTPUT(N0,N1,N2,NET_IN,W,I,O);
      --Also computes intermediate outputs of neurons
      --throughout the network.

      if NODE_BEL = 0 then
         SUM:= SUM - (O(3,1)-DESIRED_OUT)*O(3,1)
               *(1.0-O(3,1))*W(3,1,NODE)*I(3,1,NODE)
               *(1.0-I(3,1,NODE));
      else
         SUM:= SUM + (O(3,1)-DESIRED_OUT)*O(3,1)
               *(1.0-O(3,1))*W(3,1,NODE)*I(3,1,NODE)
               *(1.0-I(3,1,NODE))*I(2,NODE,NODE_BEL);
      end if;
   end loop;
   VALUE:= 2.0 * SUM;
end PART_DER_LYR2;

procedure PART_DER_LYR1(NODE: in positive;
                        NODE_BEL: in natural;
                        N0,N1,N2,NBR_TVS: in positive;
                        X: in TS_VALUES;
                        NET_IN: in out NET_INPUT_TYPE;
                        W: in WEIGHT_TYPE;
                        I: in out INPUT_TYPE;
                        O: in out OUTPUT_TYPE;
                        DESIRED_OUT: in out float;
                        VALUE: out float) is
--Finds VALUE of partial derivative of error function wrt
--a particular layer 1 weight W(1,NODE,NODE_BEL)
--at the point W in weight space.

   SUM1      :float:=0.0;
   SUM2,SUM3 :float;
begin
   for I1 in 1..NBR_TVS loop
      GET_TRNG_VECTOR(I1,N0,X,NET_IN,DESIRED_OUT);
```

```
        COMPUTE_NETWORK_OUTPUT(N0,N1,N2,NET_IN,W,I,O);

        --Also computes intermediate outputs of neurons
        --throughout the network.

        SUM2:= 0.0;
        for I2 in 1..N2 loop
           if NODE_BEL = 0 then
              SUM3:= -W(2,I2,NODE)*I(2,I2,NODE)
                     *(1.0-I(2,I2,NODE));
           else
              SUM3:= 0.0;
              for I3 in 1..N1 loop
                 SUM3:= SUM3 + W(2,I2,I3)*I(2,I2,I3)
                      *(1.0-I(2,I2,I3))*I(1,NODE,NODE_BEL);
              end loop;
           end if;
           SUM2:= SUM2 + W(3,1,I2)*I(3,1,I2)
                  *(1.0-I(3,1,I2))*SUM3;
        end loop;
        SUM1:= SUM1 + (O(3,1)-DESIRED_OUT)*O(3,1)
               *(1.0-O(3,1))*SUM2;
     end loop;
     VALUE:= 2.0 * SUM1;
  end PART_DER_LYR1;

  procedure FIND_P1(N0,N1,N2,NBR_TVS: in positive;
                    X: in TS_VALUES;
                    NET_IN: in out NET_INPUT_TYPE;
                    W,W1: in out WEIGHT_TYPE;
                    I: in out INPUT_TYPE;
                    O: in out OUTPUT_TYPE;
                    DESIRED_OUT,VALUE: in out float) is
  --Finds weight space coords P1 of 1st update estimate.
  --Weights are updated from the top layer (first) to the
  --bottom layer (last).

  begin
     for C1 in 0..N2 loop
        PART_DER_LYR3(C1,N0,N1,N2,NBR_TVS,X,NET_IN,W,I,O,
                      DESIRED_OUT,VALUE);
        W1(3,1,C1):= W(3,1,C1) - A1*VALUE; --VALUE of der
        W(3,1,C1):= W1(3,1,C1); --sets wts for use by
                                --PART_DER_LYR2 and
                                --PART_DER_LYR1
     end loop;
     for C1 in 1..N2 loop
        for C2 in 0..N1 loop
           PART_DER_LYR2(C1,C2,N0,N1,N2,NBR_TVS,X,NET_IN,
                         W,I,O,DESIRED_OUT,VALUE);
           W1(2,C1,C2):=W(2,C1,C2)-A1*VALUE; --VALUE of der
           W(2,C1,C2):= W1(2,C1,C2); --sets wts for LYR1
                                     --use by PART_DER_LYR1
```

```
      end loop;
   end loop;
   for C1 in 1..N1 loop
      for C2 in 0..N0 loop
         PART_DER_LYR1(C1,C2,N0,N1,N2,NBR_TVS,X,NET_IN,
                       W,I,O,DESIRED_OUT,VALUE);
         W1(1,C1,C2):=W(1,C1,C2)-A1*VALUE; --VALUE of der
         W(1,C1,C2):= W1(1,C1,C2);
      end loop;
   end loop;
end FIND_P1;

procedure FIND_P2(N0,N1,N2,NBR_TVS: in positive;
                  X: in TS_VALUES;
                  NET_IN: in out NET_INPUT_TYPE;
                  W,W2: in out WEIGHT_TYPE;
                  I: in out INPUT_TYPE;
                  O: in out OUTPUT_TYPE;
                  DESIRED_OUT,VALUE: in out float) is
--Finds weight space coords P2 of 2nd update estimate.
--Weights are updated from the top layer (first) to the
--bottom layer (last).

begin
   for C1 in 0..N2 loop
      PART_DER_LYR3(C1,N0,N1,N2,NBR_TVS,X,NET_IN,W,I,O,
                    DESIRED_OUT,VALUE);
      W2(3,1,C1):= W(3,1,C1) - A1*VALUE; --VALUE of der
      W(3,1,C1):= W2(3,1,C1); --sets wts for use by
                              --PART_DER_LYR2 and
                              --PART_DER_LYR1
   end loop;
   for C1 in 1..N2 loop
      for C2 in 0..N1 loop
         PART_DER_LYR2(C1,C2,N0,N1,N2,NBR_TVS,X,NET_IN,
                       W,I,O,DESIRED_OUT,VALUE);
         W2(2,C1,C2):=W(2,C1,C2)-A2*VALUE; --VALUE of der
         W(2,C1,C2):= W2(2,C1,C2); --sets wts for LYR1
                                   --use by PART_DER_LYR1
      end loop;
   end loop;
   for C1 in 1..N1 loop
      for C2 in 0..N0 loop
         PART_DER_LYR1(C1,C2,N0,N1,N2,NBR_TVS,X,NET_IN,
                       W,I,O,DESIRED_OUT,VALUE);
         W2(1,C1,C2):=W(1,C1,C2)-A2*VALUE; --VALUE of der
         W(1,C1,C2):= W2(1,C1,C2);
      end loop;
   end loop;
end FIND_P2;

procedure MOVE_WT_VALS(N0,N1,N2: in positive;
                       WTS1: in WEIGHT_TYPE;
```

```
                          WTS2: out WEIGHT_TYPE) is
--In procedure TRAIN, it is sometimes necessary to move
--weight space coordinates to an array with a different
--name.  This proc. moves the weights in WTS1 to WTS2.

begin
  for C1 in 0..N2 loop
     WTS2(3,1,C1):= WTS1(3,1,C1);
  end loop;
  for C1 in 1..N2 loop
      for C2 in 0..N1 loop
         WTS2(2,C1,C2):= WTS1(2,C1,C2);
      end loop;
   end loop;
   for C1 in 1..N1 loop
      for C2 in 0..N0 loop
         WTS2(1,C1,C2):= WTS1(1,C1,C2);
      end loop;
   end loop;
end MOVE_WT_VALS;

procedure COMPUTE_GRAD_SQ(N0,N1,N2,NBR_TVS: in positive;
                          X: in TS_VALUES;
                          NET_IN: in out NET_INPUT_TYPE;
                          W: in WEIGHT_TYPE;
                          I: in out INPUT_TYPE;
                          O: in out OUTPUT_TYPE;
                          DESIRED_OUT,VALUE:in out float;
                          GRAD_SQ: out float) is
--Finds square of gradient GRAD_SQ at point P0 by summing
--the squares of all partial derivatives.

   TEMP:float:=0.0;
begin
   for C1 in 0..N2 loop
      PART_DER_LYR3(C1,N0,N1,N2,NBR_TVS,X,NET_IN,W,I,O,
                    DESIRED_OUT,VALUE);
      TEMP:= TEMP + VALUE**2;
   end loop;
   for C1 in 1..N2 loop
      for C2 in 0..N1 loop
         PART_DER_LYR2(C1,C2,N0,N1,N2,NBR_TVS,X,NET_IN,
                       W,I,O,DESIRED_OUT,VALUE);
         TEMP:= TEMP + VALUE**2;
      end loop;
   end loop;
   for C1 in 1..N1 loop
      for C2 in 0..N0 loop
         PART_DER_LYR1(C1,C2,N0,N1,N2,NBR_TVS,X,NET_IN,
                       W,I,O,DESIRED_OUT,VALUE);
         TEMP:= TEMP + VALUE**2;
      end loop;
      GRAD_SQ:= TEMP;
```

```
   end loop;
end COMPUTE_GRAD_SQ;

function DIST_TO_PT(N0,N1,N2: in positive;
                    W,WTS:in WEIGHT_TYPE) return float is
--Finds Euclidean distance from P0 (W) to P1 or P2 (WTS).

   TEMP:float:=0.0;
begin
   for C1 in 0..N2 loop
      TEMP:= TEMP + (WTS(3,1,C1) - W(3,1,C1))**2;
   end loop;
   for C1 in 1..N2 loop
      for C2 in 0..N1 loop
         TEMP:=TEMP+(WTS(2,C1,C2)-W(2,C1,C2))**2;
      end loop;
   end loop;
   for C1 in 1..N1 loop
      for C2 in 0..N0 loop
         TEMP:=TEMP+(WTS(1,C1,C2)-W(1,C1,C2))**2;
      end loop;
   end loop;
   return sqrt(TEMP);
end DIST_TO_PT;

procedure TRAIN(N0,N1,N2,NBR_TVS: in positive;
                X: in TS_VALUES;
                W,W1,W2,WH: in out WEIGHT_TYPE;
                NET_IN: in out NET_INPUT_TYPE;
                I: in out INPUT_TYPE;
                O: in out OUTPUT_TYPE;
                DESIRED_OUT,VALUE,GRAD_SQ,ERROR: in out
                                                 float) is
--Starts at initial point P0 from procedure SET_INIT_WTS
--and ends with trained weights in W array.

   OUTFILE: text_io.file_type;
begin
   create(OUTFILE,out_file,"TRNDATA");
   put_line(OUTFILE,"W(2,1,1) at steps of 400 iters");
   while IT_CT < MAX_ITERS loop
      if IT_CT mod 400 = 0 then
         put(OUTFILE,W(2,1,1)); new_line(OUTFILE);
      end if;
      --Will result in every 400th update of W(2,1,1)
      --being output to show typical weight updating
      COMPUTE_ERROR(N0,N1,N2,NBR_TVS,X,W,NET_IN,I,O,
                    DESIRED_OUT,ERROR);
      --Computes network ERROR based on all training
      --vectors at the point P0 corresponding to the
      --current point W in weight space.

      E0:=ERROR;
```

```
if IT_CT = 0 then
   E0_1ST:= E0;
end if;
--E0_1ST is the initial error function value.  It's
--extracted here and later output in file TRNDATA
--for convenience in monitoring network performance

MOVE_WT_VALS(N0,N1,N2,W,WH); --from W array to
                             --WH array
FIND_P1(N0,N1,N2,NBR_TVS,X,NET_IN,W,W1,I,O,
        DESIRED_OUT,VALUE);
--Finds all P1 coords. and moves them into W array

COMPUTE_ERROR(N0,N1,N2,NBR_TVS,X,W,NET_IN,I,O,
              DESIRED_OUT,ERROR);
--Computes network ERROR based on all training
--vectors at the point P1 corresponding to the
--current point W in weight space.

E1:=ERROR;
MOVE_WT_VALS(N0,N1,N2,WH,W); --from WH array to
                             --W array
COMPUTE_GRAD_SQ(N0,N1,N2,NBR_TVS,X,NET_IN,W,I,O,
                DESIRED_OUT,VALUE,GRAD_SQ);
--Finds the value GRAD_SQ of the square of the
--gradient of the error surface at the point P0.

if GRAD_SQ = 0.0 then exit;
--In this case, P0 is at least a local extremum.

end if;
A2D:= 2.0 * (E1 - E0 + GRAD_SQ * A1);
if A2D = 0.0 then
   A2:= A1;
--In the unlikely case that E1 - E0 + GRAD_SQ * A1
-- = 0, necessarily E1 < E0.  A2:= A1 makes P1 = P2
--and the next minimum is E1.

else
   A2:= GRAD_SQ * (A1**2)/A2D;
--This A2 is the most commonly used value for the
--step size to the extremum of the quadratic
--approximation to the error surface cross section.

end if;
MOVE_WT_VALS(N0,N1,N2,W,WH); --from W array to
                             --WH array
FIND_P2(N0,N1,N2,NBR_TVS,X,NET_IN,W,W2,I,O,
        DESIRED_OUT,VALUE);
--Finds all P2 coords. and moves them into W array

COMPUTE_ERROR(N0,N1,N2,NBR_TVS,X,W,NET_IN,I,O,
              DESIRED_OUT,ERROR);
```

```
      --Computes network ERROR based on all training
      --vectors at the point P2 corresponding to the
      --current point W in weight space.

      E2:=ERROR;
      MOVE_WT_VALS(N0,N1,N2,WH,W); --from WH array to
                                   --W array
      if (E1<=E0 and E1<=E2) and
         DIST_TO_PT(N0,N1,N2,W,W1) > WTS_CLOSE
         then MOVE_WT_VALS(N0,N1,N2,W1,W);
      --In this case, E1 is the minimum value and P1
      --(stored in W1) is far from P0 (stored in W).
      --MOVE_WT_VALS moves W1 to W so that P1 becomes
      --P0 in the next iteration.

      elsif (E2<=E0 and E2<=E1) and
            DIST_TO_PT(N0,N1,N2,W,W2) > WTS_CLOSE
            then MOVE_WT_VALS(N0,N1,N2,W2,W);
      --In this case, E2 is the minimum and P2 (stored in
      --W2) is far from P0 (stored in W).  MOVE_WT_VALS
      --moves W2 to W so that P2 becomes P0 in the next
      --iteration.

      else
      --Now either E0 is minimum or weights are close.
      --Notice that too large an initial step size A1
      --could result in the undesirable condition where
      --E0 is not close to an error surface minimum yet
      --still E0 < E1 and E0 < E2.
         exit;
      end if;
      IT_CT:= IT_CT + 1;
   end loop;

   --Provide remarks in file named TRNDATA to monitor
   --training:
   put_line(OUTFILE,"Final value of W(2,1,1) was");
   put(OUTFILE,W(2,1,1)); new_line(OUTFILE);
   put_line(OUTFILE,"Nbr of wt. updates performed was");
   put(OUTFILE,IT_CT); new_line(OUTFILE);
   put_line(OUTFILE,"Final value of W(1,1,0) was");
   put(OUTFILE,W(1,1,0)); new_line(OUTFILE);
   put_line(OUTFILE,"Final value of W(1,N1,N0) was");
   put(OUTFILE,W(1,N1,N0)); new_line(OUTFILE);
   put(OUTFILE,"Distance from P0 to P1 was");
   new_line(OUTFILE);
   put(OUTFILE,DIST_TO_PT(N0,N1,N2,WH,W1));
   new_line(OUTFILE);
   put(OUTFILE,"Distance from P0 to P2 was");
   new_line(OUTFILE);
   put(OUTFILE,DIST_TO_PT(N0,N1,N2,WH,W2));
   new_line(OUTFILE);
   put(OUTFILE,"Original error value was");
```

```
            new_line(OUTFILE);
            put(OUTFILE,E0_1ST); new_line(OUTFILE);
            put(OUTFILE,"Final error value was");
            new_line(OUTFILE);
            put(OUTFILE,E0);
            close(OUTFILE);
        end TRAIN;
    end TRAINING_PACKAGE;
```

```
with NEURON_PACKAGE;        use NEURON_PACKAGE;
with NETWORK_PACKAGE;       use NETWORK_PACKAGE;

package ERR_SURF_PACKAGE is
   procedure GET_TRNG_VECTOR(TV_NBR,N0: in positive;
                             X: in TS_VALUES;
                             NET_IN: out NET_INPUT_TYPE;
                             DESIRED_OUT: out float);
   procedure COMPUTE_ERROR(N0,N1,N2,NBR_TVS: in positive;
                           X: in TS_VALUES;
                           W: in WEIGHT_TYPE;
                           NET_IN: in out NET_INPUT_TYPE;
                           I: in out INPUT_TYPE;
                           O: in out OUTPUT_TYPE;
                           DESIRED_OUT: in out float;
                           ERROR: out float);
end ERR_SURF_PACKAGE;
```

```
with system;
with float_math_lib;          use float_math_lib;
with math_lib_extension;      use math_lib_extension;

package body ERR_SURF_PACKAGE is

   procedure GET_TRNG_VECTOR(TV_NBR,N0: in positive;
                             X: in TS_VALUES;
                             NET_IN: out NET_INPUT_TYPE;
                             DESIRED_OUT: out float) is
   --Assigns the first N0 components of training vector
   --number TV_NBR sequentially to the network inputs, and
   --assigns the last component to the variable DESIRED_OUT.
   --Since all time series values fall between 0 and 2, it
   --divides each component by 2 to enforce a normalized
   --range of 0 to 1 (the sigmoidal network output is always
   --in this range).

   begin
      for C1 in 1..N0 loop
         NET_IN(C1):= X(TV_NBR + C1 - 2)/2.0;
      end loop;
      DESIRED_OUT:= X(TV_NBR + N0 - 1)/2.0;
   end GET_TRNG_VECTOR;

   procedure COMPUTE_ERROR(N0,N1,N2,NBR_TVS: in positive;
                           X: in TS_VALUES;
                           W: in WEIGHT_TYPE;
                           NET_IN: in out NET_INPUT_TYPE;
                           I: in out INPUT_TYPE;
                           O: in out OUTPUT_TYPE;
                           DESIRED_OUT: in out float;
                           ERROR: out float) is
   --Computes the network ERROR at point W based on all
   --training vectors.

      SUM      :float:= 0.0;
      SQ_TERM  :float;
   begin
      for C1 in 1..NBR_TVS loop
         GET_TRNG_VECTOR(C1,N0,X,NET_IN,DESIRED_OUT);
         COMPUTE_NETWORK_OUTPUT(N0,N1,N2,NET_IN,W,I,O);
         SQ_TERM:= (O(3,1) - DESIRED_OUT)**2;
         SUM:= SUM + SQ_TERM;
      end loop;
      ERROR:= SUM;
   end COMPUTE_ERROR;

end ERR_SURF_PACKAGE;
```

```
with NEURON_PACKAGE;          use NEURON_PACKAGE;

package NETWORK_PACKAGE is
   type TS_VALUES is array(natural range <>) of float;
   type NET_INPUT_TYPE is array(positive range <>) of float;
   procedure GET_PARAMS(N0,N1,N2,NBR_TVS,NBR_PVS: out
                                                  positive);
   procedure SET_INIT_WTS(N0,N1,N2: in positive;
                          W: out WEIGHT_TYPE);
   procedure GET_TS_VALUES(N0,NBR_TVS,NBR_PVS: in positive;
                           X: in out TS_VALUES;
                           X_PRED: out TS_VALUES);
   function NEURON_OUTPUT(N0,N1,N2,LAYER,NODE: in positive;
                          W: in WEIGHT_TYPE;
                          I: in INPUT_TYPE) return float;
   procedure COMPUTE_NETWORK_OUTPUT(N0,N1,N2: in positive;
                                    NET_IN: in
                                            NET_INPUT_TYPE;
                                    W: in WEIGHT_TYPE;
                                    I: in out INPUT_TYPE;
                                    O: out OUTPUT_TYPE);
end NETWORK_PACKAGE;
```

```
with system;
with text_io;                  use text_io;
with integer_text_io;          use integer_text_io;
with float_text_io;            use float_text_io;
with float_math_lib;           use float_math_lib;
with math_lib_extension;       use math_lib_extension;

package body NETWORK_PACKAGE is

   procedure GET_PARAMS(N0,N1,N2,
                        NBR_TVS,NBR_PVS: out positive) is
   --Gets five constant positive integers from the input
   --file PARAMS.  N0, N1, and N2 are the numbers of neurons
   --in the input, first, and second network layers,
   --respectively.  NBR_TVS is the number of training
   --vectors used, and NBR_PVS is the number of time series
   --values to be predicted.

      INFILE1: text_io.file_type;
   begin
      open(INFILE1,in_file,"PARAMS");
      get(INFILE1,N0); skip_line(INFILE1);
      get(INFILE1,N1); skip_line(INFILE1);
      get(INFILE1,N2); skip_line(INFILE1);
      get(INFILE1,NBR_TVS); skip_line(INFILE1);
      get(INFILE1,NBR_PVS); skip_line(INFILE1);
      close(INFILE1);
   end GET_PARAMS;

   procedure SET_INIT_WTS(N0,N1,N2: in positive;
                          W: out WEIGHT_TYPE) is
   --Sets initial weights to values near zero.

      seed     :system.unsigned_longword:= get_seed;
      u_center :constant:=0.0;
      u_width  :constant:=1.6;
   begin
      for C1 in 1..N1 loop  --set LYR1 init weights and
                            --thresholds to values within
                            --u_width of 0
         for C0 in 0..N0 loop
            uniform(u_center,u_width,seed,W(1,C1,C0));
         end loop;
      end loop;
      for C2 in 1..N2 loop  --set LYR2 init weights and
                            --thresholds to values within
                            --u_width of 0
         for C1 in 0..N1 loop
            uniform(u_center,u_width,seed,W(2,C2,C1));
         end loop;
      end loop;
      for C1 in 0..N2 loop  --set LYR3 init weights and
                            --threshold to values within
```

```
                              --u_width of 0
      uniform(u_center,u_width,seed,W(3,1,C1));
   end loop;
end SET_INIT_WTS;

procedure GET_TS_VALUES(N0,NBR_TVS,NBR_PVS: in positive;
                        X: in out TS_VALUES;
                        X_PRED: out TS_VALUES) is
--Gets all time series values used from the input file
--GMCDATA and puts them in the array X.  Those values
--which the network does not predict are also put into
--the array X_PRED since the prediction process does
--initially use some actual time series values.

   INFILE2: text_io.file_type;
begin
   open(INFILE2,in_file,"GMCDATA");
   for C1 in 0..N0 + NBR_TVS + NBR_PVS - 1 loop
      get(INFILE2,X(C1));
      skip_line(INFILE2);
   end loop;
   for C1 in 0..N0 + NBR_TVS - 1 loop
      X_PRED(C1):= X(C1);
   end loop;
   close(INFILE2);
end GET_TS_VALUES;

function NEURON_OUTPUT(N0,N1,N2,LAYER,NODE: in positive;
                       W: in WEIGHT_TYPE;
                       I: in INPUT_TYPE) return float is
--Computes the OUT_PUT of the neuron at NODE in LAYER.

   SUM:float:=0.0;
   END_OF_LAYER:positive;
   OUT_PUT:float;
   TEMP:float;
begin
   if LAYER = 1 then
      END_OF_LAYER:= N0;
   elsif LAYER = 2 then
      END_OF_LAYER:= N1;
   else
      END_OF_LAYER:= N2;
   end if;
   for J in 1..END_OF_LAYER loop
      TEMP:= W(LAYER,NODE,J) * I(LAYER,NODE,J);
      SUM:= SUM + TEMP;
   end loop;
   OUT_PUT:= 1.0/(1.0+exp(-SUM + W(LAYER,NODE,0)));
   return OUT_PUT;
end NEURON_OUTPUT;

procedure COMPUTE_NETWORK_OUTPUT(N0,N1,N2: in positive;
```

```
                                        NET_IN: in
                                                NET_INPUT_TYPE;
                                        W: in WEIGHT_TYPE;
                                        I: in out INPUT_TYPE;
                                        O: out OUTPUT_TYPE) is
      --Finds the output O(3,1) of the network given a
      --particular set of network inputs NET_IN, and in the
      --process also finds interconnecting inputs.

      begin
         --set first layer inputs
         for C1 in 1..N1 loop
            for C2 in 1..N0 loop
               I(1,C1,C2):= NET_IN(C2);
            end loop;
         end loop;
         --find inputs to layer 2 nodes
         for CI in 1..N2 loop
            for CJ in 1..N1 loop
               I(2,CI,CJ):= NEURON_OUTPUT(N0,N1,N2,1,CJ,W,I);
            end loop;
         end loop;
         --find inputs to layer 3 nodes
         for CI in 1..N2 loop
            I(3,1,CI):= NEURON_OUTPUT(N0,N1,N2,2,CI,W,I);
         end loop;
         --find network output
         O(3,1):= NEURON_OUTPUT(N0,N1,N2,3,1,W,I);
      end COMPUTE_NETWORK_OUTPUT;

   end NETWORK_PACKAGE;
```

```
package NEURON_PACKAGE is
   --NEURON_PACKAGE has only a specification part.  It
   --defines the ranges of the indices of the three neuron
   --parameters as explained in the comments included in
   --the main procedure JIMSPNET.

   type WEIGHT_TYPE is array(positive range <
                             positive range <  ,
                             natural range <>, of float;
   type OUTPUT_TYPE is array(positive range <>,
                             positive range <>) of float;
   type INPUT_TYPE is array(positive range <>,
                            positive range <>,
                            positive range <>) of float;
end NEURON_PACKAGE;
```

Appendix D: Additional Error Surface Descents

Figures 3.11 through 3.14 in Section 3.4.1 depict the performances of four weight updating algorithms. Each of these depictions begins an error surface descent from the point $(W_1, W_2) = (4, -3.5)$. This appendix shows the same four algorithms descending from the initial points (4,-4) (in Figures D.1 through D.4) and (6,-8) (in Figures D.5 through D.8). Inspection of these plots reveals the advantage of the QSD algorithm (with batch processing, unless otherwise noted) - faster convergence to lower final error values.

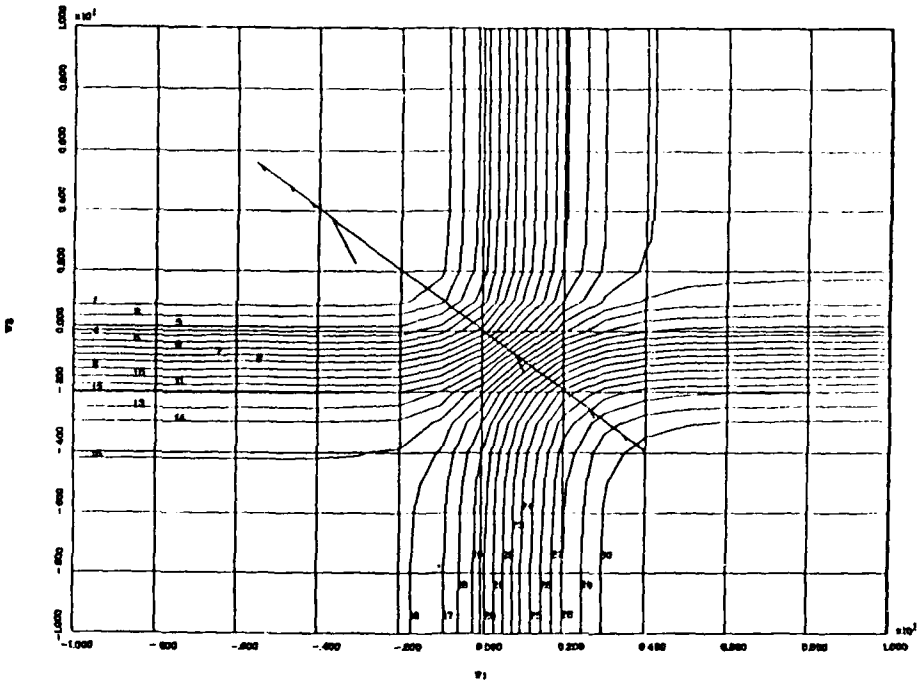


Figure D.1 QSD Algorithm from (4,-4)

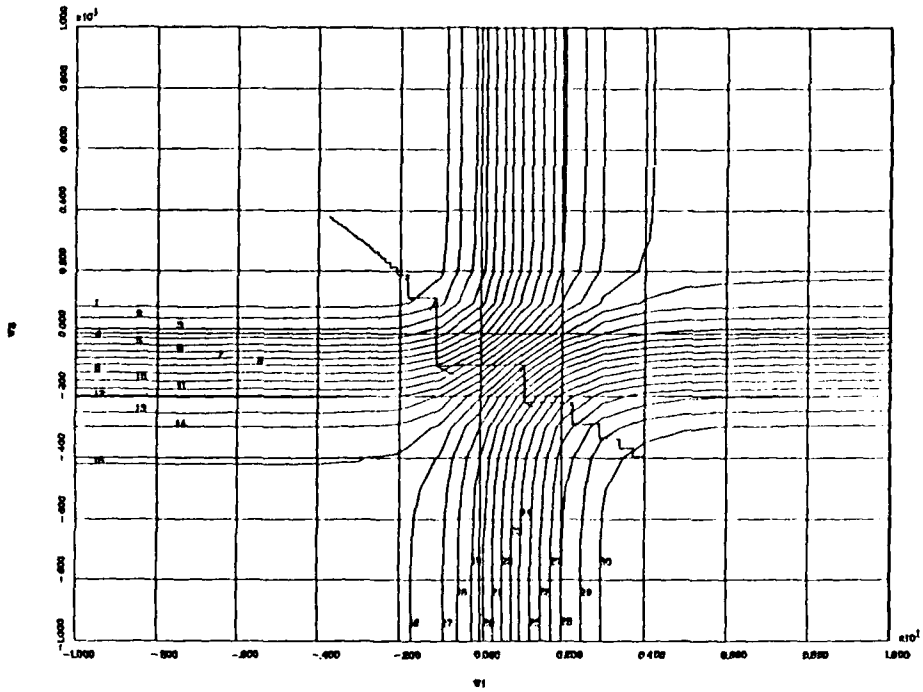


Figure D.2 Single-Step, Input-by-Input Updating from (4,-4)

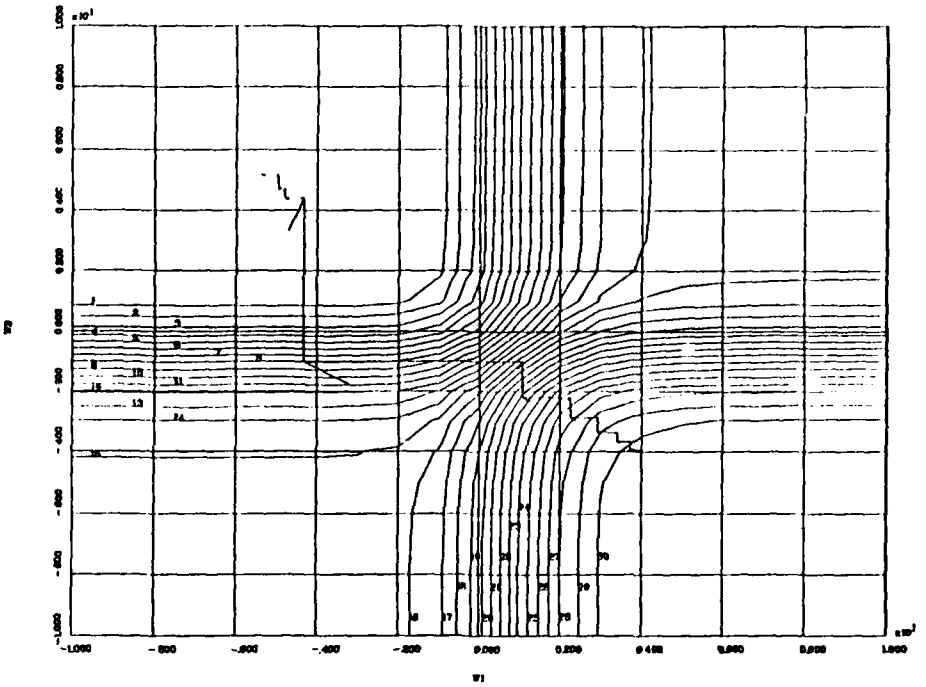


Figure D.3 Input-by-Input Application of QSD from (4,-4)

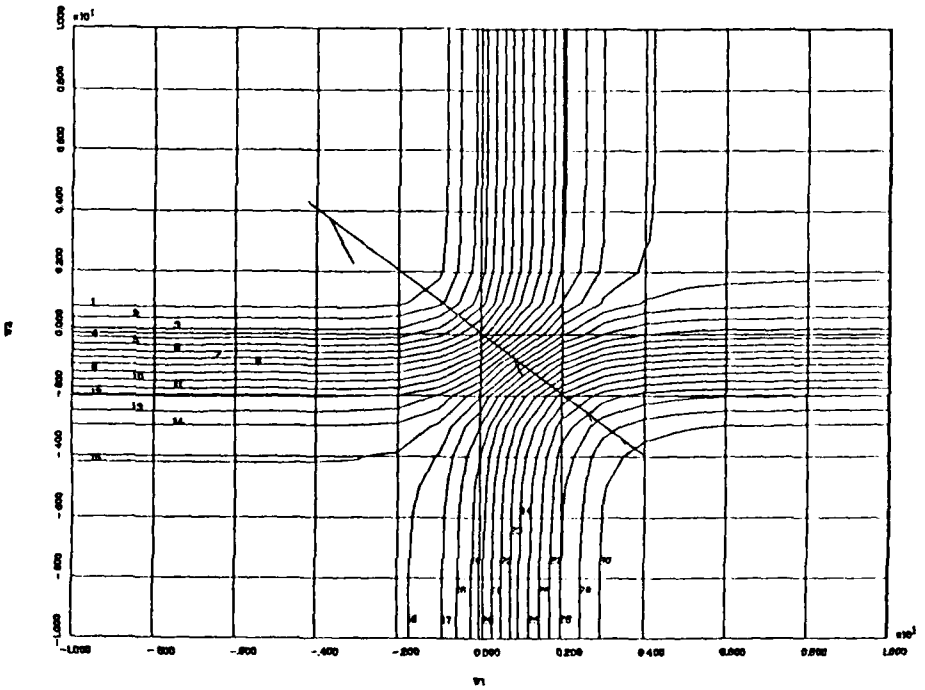


Figure D.4 Single-Step Batch Processing from (4,-4)

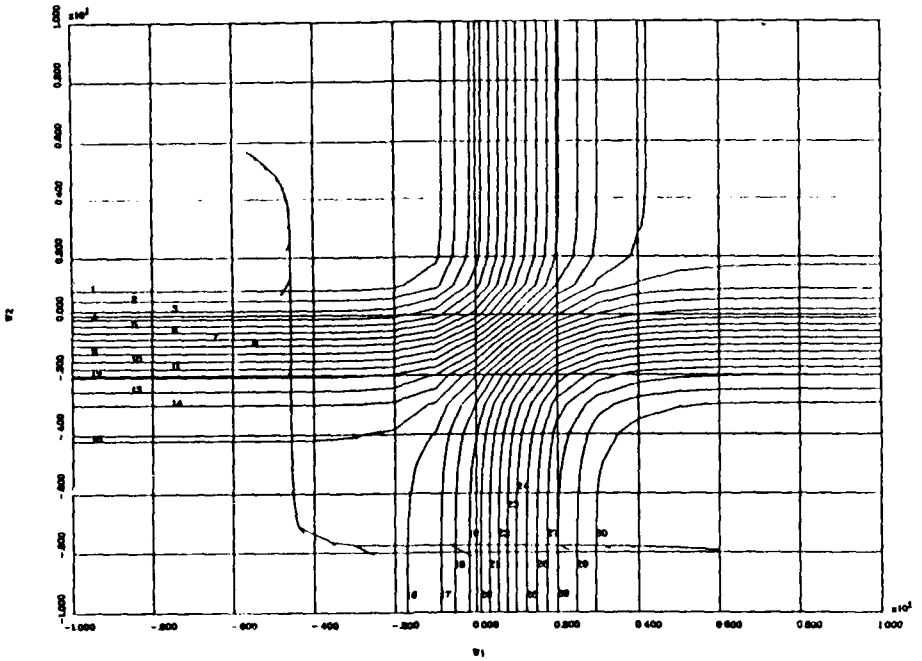


Figure D.5 QSD Algorithm from (6,-8)

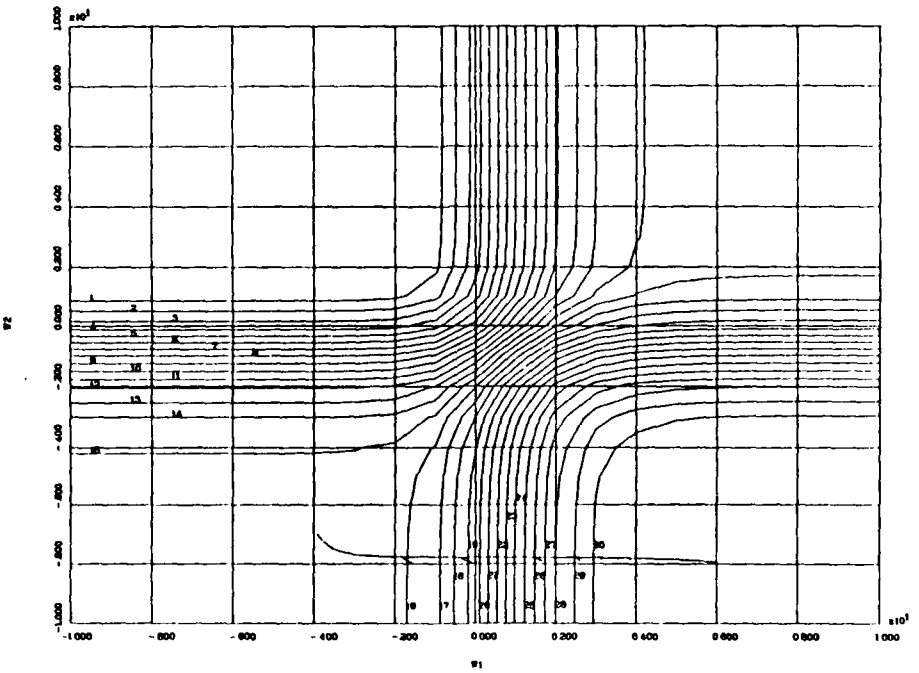


Figure D.6 Single-Step, Input-by-Input Updating from (6,-8)

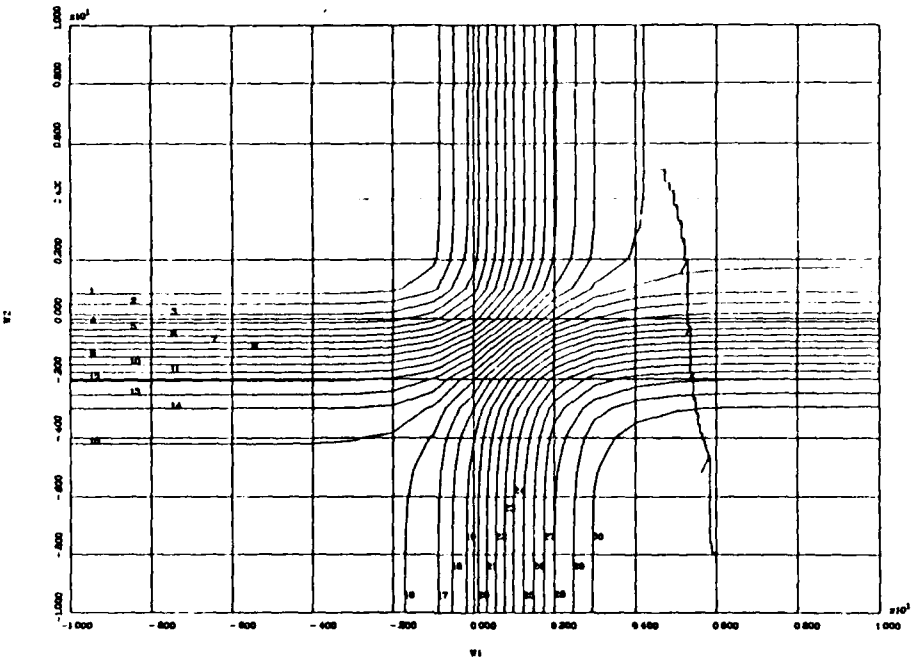


Figure D.7 Input-by-Input Application of QSD from (6,-8)

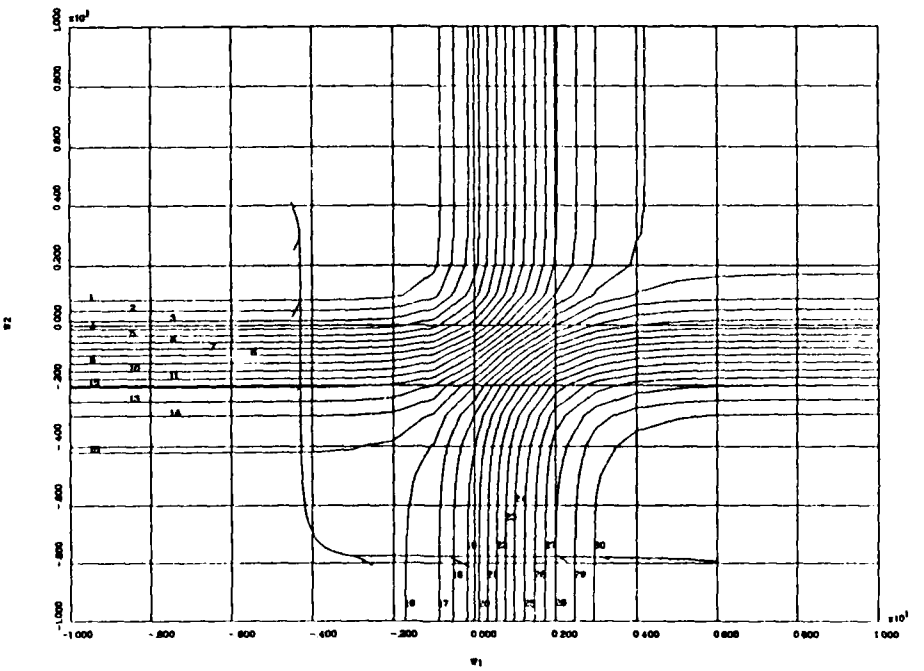


Figure D.8 Single-Step Batch Processing from (6,-8)

Bibliography

Booch, Grady. Software Engineering with Ada. Menlo Park CA: The Benjamin/Cummings Publishing Company, 1983.

Conte, S. D. and Carl de Boor. Elementary Numerical Analysis: An Algorithmic Approach (Second Edition). New York: McGraw-Hill Book Company, 1972.

Dahl, Edward D. "Accelerated Learning Using the Generalized Delta Rule," Proceedings of the IEEE International Conference on Neural Networks, 2. 523-530. San Diego: SOS Printing, 1987.

Farmer, J. Doyne. "Chaotic Attractors of an Infinite-Dimensional Dynamical System," Physica D, 4: 366-393 (1982).

Farmer, J. Doyne and John J. Sidorowich. Exploiting Chaos to Predict the Future and Reduce Noise, Version 1.1. Report LA-UR-88-901. Los Alamos NM: Los Alamos National Laboratory, February 1988.

Feynman, Richard P. The Character of Physical Law. Cambridge, Mass.: M.I.T. Press, 1965.

Froehling, Harold and others. "On Determining the Dimension of Chaotic Flows," Physica D, 3: 605-617 (1981).

Gleick, James. Chaos: Making a New Science. New York: Viking Penguin, 1988.

Grassberger, Peter and Itamar Procaccia. "Measuring the Strangeness of Strange Attractors," Physica D, 9: 189-208 (1983).

Lapedes, Alan and Robert Farber. How Neural Nets Work, preprint. Report LA-UR-88-418. Los Alamos NM: Los Alamos National Laboratory, 1988a. Submitted to Proc. of IEEE Conf. on Neural Info. Processing Systems.

Lapedes, Alan and Robert Farber. Nonlinear Signal Processing Using Neural Networks: Prediction and System Modeling, preprint. Report LA-UR-87-2662. Los Alamos NM: Los Alamos National Laboratory, 1988b. Submitted to Biological Cybernetics.

Lippmann, Richard P. "An Introduction to Computing with Neural Nets," IEEE ASSP Magazine: 4-22 (April 1987).

Mackey, Michael C. and Leon Glass. "Oscillation and Chaos in Physiological Control Systems," Science, 197: 287-289 (15 July 1977).

Maybeck, Peter S. Class notes distributed in EENG768, Computational Aspects of Modern Control, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, Fall 1988.

Ruck, Capt Dennis W. Multisensor Target Detection and Classification. MS thesis, AFIT/GE/ENG/87D-56. School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB Ohio, December 1987.

Tarr, Capt Gregory L. Dynamic Analysis of Feed Forward Neural Networks Using Simulated and Measured Data. MS thesis, in preparation. School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB Ohio, December 1988.

Thompson, J.M.T. and H.B. Stewart. Nonlinear Dynamics and Chaos. Chichester, England: John Wiley and Sons, 1986.

VITA

Captain James R. Stright [redacted] He graduated from [redacted] school in Meadville, Pennsylvania, and from Allegheny College in Meadville, where he majored in mathematics. He joined the Air Force in 1983, and earned a bachelor's degree in electrical engineering from Gannon University in Erie, Pennsylvania in 1984. Following graduation from OTS in August 1984, his first assignment was to the National Computer Security Center at Fort Meade, Maryland, where he served as the government team leader for development of an encryption device. He entered the School of Engineering, Air Force Institute of Technology, in June 1987.

[redacted]

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GE/ENG/88D-50	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION School of Engineering 6b. OFFICE SYMBOL (If applicable) AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code) Air Force Institute of Technology Wright-Patterson AFB OH 45433-6583	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION RADC Griffiss AFB NY 13441 8b. OFFICE SYMBOL (If applicable) COTC	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification) See Box 19	
12. PERSONAL AUTHOR(S) James R. Stright, B.E.E., Capt, USAF	
13a. TYPE OF REPORT MS Thesis 13b. TIME COVERED FROM _____ TO _____ 14. DATE OF REPORT (Year, Month, Day) 1988 December 15. PAGE COUNT 120	
16. SUPPLEMENTARY NOTATION	

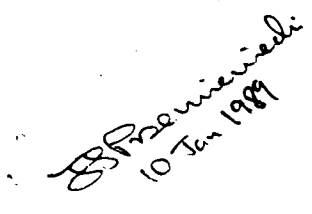
17. COSATI CODES		
FIELD	GROUP	SUB-GROUP
06	04	

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Neural nets, Predictions, Time Series Analysis

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Title: A NEURAL NETWORK IMPLEMENTATION OF
CHAOTIC TIME SERIES PREDICTION

Thesis Chairman: Steven K. Rogers, Captain, USAF
Associate Professor of Electrical Engineering



20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL James R. Stright, Captain, USAF	22b TELEPHONE (Include Area Code) (513) 255-3030 22c. OFFICE SYMBOL AFIT/ENG

DD Form 1473, JUN 86 *Previous editions are obsolete.*

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

19. Con't

Abstract

This thesis provides a description of how a neural network can be trained to "learn" the order inherent in chaotic time series data and then use that knowledge to predict future time series values. It examines the meaning of chaotic time series data, and explores in detail the Glass-Mackey nonlinear differential delay equation as a typical source of such data. An efficient weight update algorithm is derived, and its two-dimensional performance is examined graphically. A predictor network which incorporates this algorithm is constructed and used to predict chaotic data.

The network was able to predict chaotic data. Prediction was more accurate for data having a low fractal dimension than for high-dimensional data. Lengthy computer run times were found essential for adequate network training.